Sao-Jie Chen
Guang-Huei Lin
Pao-Ann Hsiung
Yu-Hen Hu

Hardware Software Co-Design of a Multimedia SOC Platform

Foreword by
Giovanni de Micheli

Springer

Hardware Software Co-Design of a Multimedia SOC Platform

Hardware Software Co-Design of a Multimedia SOC Platform

by

Sao-Jie Chen
National Taiwan University, Taipei, Taiwan, ROC

Guang-Huei Lin
National Taiwan University, Taipei, Taiwan, ROC

Pao-Ann Hsiung
National Chung Cheng University, Chia-Yi, Taiwan, ROC

and

Yu-Hen Hu
University of Wisconsin-Madison, Madison, USA

Springer

Prof. Sao-Jie Chen
National Taiwan University
Dept. Electrical Engineering
Graduate Inst. of Electronics
Engineering
Taipei 106
Taiwan R.O.C.
csj@cc.ee.ntu.edu.tw

Dr. Guang-Huei Lin
National Taiwan University
Dept. Electrical Engineering
Graduate Inst. of Electronics
Engineering
Taipei 106
Taiwan R.O.C.

Dr. Pao-Ann Hsiung
National Chung Cheng
University
Chia-Yi 621
Taiwan R.O.C.

Dr. Yu-Hen Hu
University of Wisconsin,
Madison
Dept. Electrical & Computer
Engineering
1415 Engineering Drive
Madison WI 53706
2556 Engineering Hall
USA

ISBN: 978-1-4020-9622-8 e-ISBN: 978-1-4020-9623-5

DOI 10.1007/978-1-4020-9623-5

Library of Congress Control Number: 2008941282

Printed on acid-free paper

9 8 7 6 5 4 3 2 1

springer.com

Foreword

System-level design is a key design technology to realize integrated systems of various types, possibly integrated on a single die, i.e., *Systems on Chips*, or assembled in a single package, i.e., *Systems in Package*. As technology advances and systems become increasingly more complex, the use of high-level abstractions and platform-based design becomes more and more a necessity. Indeed the productivity of designers increases with the abstraction level, as demonstrated by practices in both the software and hardware domains. The use of high-level models and platforms allow designers to be productive, even when they have weaker specific skills in circuit design technology.

Software plays a key role in embedded system design and is crucial for system programmability and flexibility. The latter factor is extremely important for extending the life of components across different families of products. Compilation is the crucial technology to achieve effective system operation on platforms, starting from high-level programming constructs. The compiler technology has evolved tremendously, especially in the domain of *application-specific instruction set processor* design which is commonplace in embedded platforms.

Effective software compilation is the companion to hardware compilation, also called high-level synthesis. Both share important objectives, like addressing real-time constraints and system performance. The software and hardware design technologies show complementarities and find their most natural applications on platform-based design.

Multi-media systems benefit largely from modular and flexible realizations relying on platforms. The market for these systems is rapidly expanding and its future hinges in part on the industrial capability to deliver advanced systems with rapid turn-around time. In this perspective, the technologies described in this book are very significant for progress of science and technology and have direct impact on applications ranging from entertainment to medical imaging, from terrestrial environmental monitoring to defense.

Lausanne
November 25, 2008

Giovanni De Micheli

Preface

This book is the outcome of a recent international collaborative research project aiming at developing both the hardware and software of a platform based SoC (System-on-Chip) architecture for embedded multimedia systems. Since its inception a decade ago, SoC has captured the attentions of application specific integrated circuit (ASIC) design houses, computer aided design (CAD) companies, and embedded system developers. In particular, the immense popularity of killer multimedia gadgets such as iPod, and iPhone has fueled unprecedented interests in developing new generation multimedia SoC systems.

However, the high level of integration also brings great challenges to system designers: Conventional component-oriented design methodologies can no longer handle the ever increasing system complexity. Hardware and software are necessarily becoming convergent and must be fully concurrent design endeavors. Hardware engineers must understand higher level signal processing algorithms, functional simulations, and design verifications. Software engineers, on the other hand, must pay great attention to heterogeneous instruction set architectures, timing and power constraints, and hardware-in-the-loop system level simulation and verifications. All these point to a diverse body of knowledge, and skills that a competent SoC designer must be equipped. Currently, such valuable knowledge sources are scattered in many different text books, research papers, and other on-line sources. For someone who is interested in gaining a comprehensive overview of issues related to the hardware and software development of multimedia SoC, a highly integrated book is unavailable.

This book is written to serve this purpose. Based on a joint research project coordinated by S. J. Chen at the National Taiwan University (NTU) in Taiwan, this book is co-authored by key participants of a project entitled: Implementation of a Multimedia SoC Platform for Scalable Power-Aware Custom Embedded Systems. In this 3-year research project, we focused on a novel SoC platform based on a Subword-Parallel Single Instruction Multiple Data (SWP-SIMD) micro-architecture. Much of the materials included in this monograph are drawn from outcomes of this research project. A distinct feature of this book is to incorporate a very diverse list of subjects and tied them together elegantly to the development of an SWP-SIMD based SoC platform. Specifically, subject areas that covered include system level design methodologies, H.264 video coding algorithms, sub-word parallel micro-architecture design, SIMD vectorized compiler technology, and real time operating systems. Obviously, with limited space, it is impossible to engage in-depth discussion of each subject area. Instead, the authors' approach is to provide sufficient detail so that readers may appreciate the significance of each topic and understand the relations of a particular subject to other topics. Plenty of references are provided so that interested readers may pursue further investigation using materials presented in this book as a stepping stone. We understand that such an ambitious goal would not be easy to reach. We hope our efforts will make some tangible contributions toward promoting the SoC platform design and applications.

This book is written for engineers who are working in integrated circuit design houses, in computer aided design tool companies, and embedded system design companies; as well as graduate students who are pursuing a career in computer engineering, multimedia system implementation and related field. We will be very happy to hear readers' feedback and comments.

Taipei, Taiwan	Sao-Jie Chen
Taipei, Taiwan	Guang-Huei Lin
Chia-Yi, Taiwan	Pao-Ann Hsiung
Madison, Wisconsin	Yu-Hen Hu

Acknowledgements

The authors are greatly thankful to the foreword writer, Professor and Director Giovanni De Micheli at the Institute of Electrical Engineering and the Integrated Systems Center at the École Polytechnique Fédérale de Lausanne, Switzerland.

The authors are also grateful to those who have made great contributions to the development of this book. The PLX instruction set architecture was originally developed by Professor Ruby Lee at the Princeton University. She has provided numerous valuable inputs to the design of the PLX core. She also generously provided us with an LCC compiler, Assembler, and Loader for the PLX architecture.

Many current and former graduate students in our research groups have made tangible contributions to portions of this book. Specifically, Ya-Nan Wen and Chia-Wen Tsai of National Taiwan University have made significant contributions to the PLX and H.264 chip implementations. Xiao-Long Wu, Cheng-Cho Jean, and Hong-Kuei Chen of National Taiwan University have contributed to the development of the SIMD compiler and profiler tools. Chun-Jen Wei of National Taiwan University contributed to the development of algebraic verification tool. Chao-Sheng Lin of National Chung-Cheng University has developed the RTOS described in this book.

Finally, we would like to express our appreciation to our family members for their encouragement and patience during the writing of this book.

Taipei, Taiwan	Sao-Jie Chen
Taipei, Taiwan	Guang-Huei Lin
Chia-Yi, Taiwan	Pao-Ann Hsiung
Madison, Wisconsin	Yu-Hen Hu

Contents

List of Figures

List of Tables

Chapter 1
Introduction

What is embedded processing? One of the simplest definitions is that embedded processing is not for general purpose. General purpose processors are the ones used in desktop PCs and servers. Development tools for desktop processor are popular, and there are millions of software developed for desktop processor. Thus, using general purpose processor can reduce time-to-market. But this solution is not optimized on some critical metrics including performance, cost, power, and size. Many embedded multimedia systems are handheld systems, such as MP3 players, PDAs and 3G phones. A single general purpose processor is unable to handle real-time functions such as communication, camera, video, audio, touch screen, TV, and GPS in time, or it will consume too much power.

Many embedded processors have worse performance on general applications, but have much better performance on some specific applications than the general purpose processors. The well-known examples are *digital signal processor* (DSP) and network processor.

Embedded system devices normally embody the functionality they implemented. In other words, they are designed to run a few codes with a predictable pattern. In contrast, applications of a general purpose system are barely known in advance. A traditional embedded system design flow is to select proper processor and peripheral device controllers, then to spend most of the effort on developing software for this system.

In 1965, Gordon Moore predicted that the number of transistors incorporated in an IC would increase twofold every year. This was really an amazing prediction that proved to be more accurate than Moore had believed. In the past few decades, the scale of IC integration has been soaring high. It started from *Small Scale Integration* (SSI) with around 100 transistors per IC in 1960s, up to *Very Large Scale Integration* (VLSI) accommodating more than 10000 transistors per IC in 1980s. There is no sign that such tendency would ever cease. In recent years, the integration scale has only slightly slowed down to a factor of two for every eighteen months. The outburst of IC complexity, as predicted by Moore's Law, is driving the current semiconductor industry to challenge another cutting edge revolution: *System-on-Chip* (SoC) with the capacity of integrating more transistors in a single chip to form an entire electronic system. This concept is feasible thanks to the very exceptional manufacturing advances that bring IC nanotechnology to fruition.

As Moore's Law continues unencumbered into the nanometer era, process geometries have shrunken to 65 nm, chips are reaching the giga-gate scale. It brings the possibility to integrate all processors and peripheral device controllers into a chip.

Traditional hardware–software partitioning is simply as: critical functions performed by specific hardware and control-oriented functions by software. Here specific hardware is defined as an *application specific integrated circuit* (ASIC) which is a special design dedicated for an application, and software means running a code on a general purpose processor. In the last century, most embedded systems need specific hardware to process multimedia applications, with the constraint of power consumption or performance. In general, specific hardware is more power-efficient than software for an application with the same performance. But specific hardware is less flexible to adapt to new features.

In SoC era, *time-in-market* becomes as important as *time-to-market*. Any new chip production needs to pay very high *non-returnable engineering* (NRE) cost even if there is only a little modification from the previous version. By the same reason, we wish that the chip can sustain longer and useable for more applications. For example, in IP-based 4G wireless communication, we would like to design a chip used for both WiMAX (*Worldwide Interoperability for Microwave Access*) and LTE (*Long Term Evolution*), while these standards are not well-defined. The key to applying a single integrated circuit to multiple applications for both time-to-market and time-in-market is *programmability*.

An *application specific instruction-set processor* (ASIP) is a software-programmable processing-element tailored for this purpose. It provides efficient and economic way for a particular application computation. An ASIP may add some multimedia operations or encryption operations into its instruction-set to improve performance with low cost-overhead.

To compete with the performance of ASIC, many parallelization techniques are adapted into ASIP. These techniques include *data level parallelism* (DLP) in a *single instruction multiple data* (SIMD) processor, *instruction level parallelism* (ILP) in a *very large instruction word*(VLIW) processor, and *thread level parallelism* (TLP) in a multi-threading processor. Armed with these parallelism mechanisms, multi-core becomes more and more feasible and popular in embedded SoCs.

Therefore, a specialized parallel compiler becomes more important to optimize an application on a specific multi-core processor. This kind of parallel compiler has not only to translate high-level programming language instructions into the target ASIP codes, but also to schedule these instructions to exploit the parallelization capability of that ASIP.

Software needs to run on its target processor. While processor and compiler are designing, software is unable to design until a prototype was developed. Without verification by software, the processor is not guaranteed to meet system constraints, thus the ASIP needs to be re-designed many times. To reduce the long cycle, both developing software as early as possible and evaluating system constraints at a higher system level become very important.

The fundamental building blocks of an SoC are its *intellectual property* (IP) cores, which are reusable hardware blocks designed to perform a particular task of a given component. An IP core could either be a programmable component like a processor, or a hardware entity with fixed behavior like an MPEG accelerator. Different IP cores are interconnected on an SoC by a communication structure, such as a shared bus or *network-on-chip* (NoC), in order to establish communication among them.

IP reuse drives the progress of system-level design. A reusable IP can be obtained from the third-party IP provider. Typically IP providers would not release RTL design. The time spent to identify a third-party IP and integrate it into the designed system places this approach at an unfavorable position compared to designing the IP in-house. A higher level *Transaction Level Modeling* (TLM) description is more feasible for IP providers to protect their design.

TLM is the current promotion methodology used for hardware/software codesign before and after hardware/software partitioning. Before partitioning, TLM could be used to create a point-to-point, addressless functional yet concurrent system model, reusing IP behaviors from application engineers. After partitioning, TLM automatically wraps the behavior in the address-mapped TLM model for embedded software functional verification.

Most embedded systems are real-time, which time constraint is critical. A real-time embedded operating system is required to serve memory allocation, peripheral I/O device system calls, inter-process communication, and priority-based task scheduling.

PLX [1], developed by Professor Ruby Lee at Princeton University, is a native *subword-parallel single instruction multiple data* (SWP-SIMD) processor [2] that supports high-performance, low-cost multimedia information processing, 3-D graphical processing and permutation instructions for security operations. This book intends to discuss many of the above mentioned hardware–software codesign issues that we encountered in designing a PLX-based embedded multimedia SoC platform.

The book contents are organized as follows. Chapter 2 introduces traditional platform-based hardware–software co-design and some multimedia algorithms that require ASIP. Chapter 3 introduces some system level design techniques that we used to design our SIMD PLX core. Chapter 4 introduces ASIP processor design techniques and parallelization methodologies, such as DLP, ILP and TLP. Chapter 5 introduces parallel compiler techniques specifically tailored for PLX. Chapter 6 describes the design of PLX processor and its virtual platform TLM modeling. Chapter 7 introduces real-time operating system OS development experience for PLX. Finally, a conclusion is drawn in Chapter 8.

Chapter 2
Design Consideration

In this chapter, we briefly describe the basic concepts of a platform-based design, system-level modeling techniques used in designing a platform, and some multimedia algorithms that require specific instruction set design in a processor core, such as PLX, a native *single-instruction multiple-data* (SIMD) core developed by Professor Ruby Lee at Princeton University [1].

2.1 Platform-Based Design

A platform is a library of components that can be assembled to generate a design. This library not only contains computational blocks that carry out the appropriate computation but also communication components that are used to interconnect the functional components.

Platform-based design changes design flow from vertically-oriented into horizontally-oriented. For information protection, a product is fully designed and manufactured in an early industrial company. The increase of electronic design complexity and the advances in technologies force designers to focus on their core competence. The pressure for reducing time-to-market of electronics products in the presence of exponentially increasing complexity has forced designers to adopt methods that favor component reuse. Furthermore, each organization that contributes a component to the final product naturally strives for a position that allows it to make continuous adjustments and accommodate last-minute engineering changes.

Each design should satisfy constraints on characteristics such as performance, cost, power consumption, and weight. It brings a choice for designer to implement a function as a hardware component or as a software code running on a programmable component.

Platform-based design is a meet-in-the-middle process, where successive refinements of specifications meet with abstractions of potential implementations that are captured in the models of the elements of the platform. Figure 2.1 shows this concept [3]. The comprehensive model includes the views of platforms from the application and the implementation architecture perspectives that meet at the vertex of the two cones.

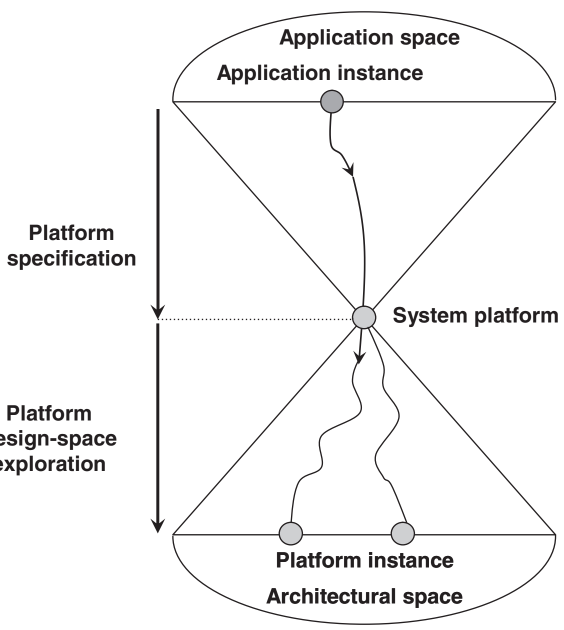


Fig. 2.1 System platform design concept

Hardware and software reuse is the key concept on platform-based design. Reusing a reference platform is the only solution to time-to-market. Table 2.1 lists some popular platforms. There are hundreds of embedded-system SoC platforms on market for different applications. Prototyping is important to verify a newly designed hardware in a platform. Some SoC platforms contains configurable field programmable gate array (FPGA) which allow users to add their hardware, such as Philip's RSP (Rapid System Prototyping) and Altera's Excalibur.

A platform should contain an architecture model and its associated design methodology. The architecture model is a predefined architecture which consists of various families of components such as processor, memory, function blocks, and system bus/communications. The design methodology is constructed as an integrated design flow with multiple levels at which component modeling, simulation environment, and in-circuit emulator (ICE) are provided.

A platform is mostly provided by a processor vendor. Today the most widely used platform for SoC is based on the ARM processor. Figure 2.2 shows an example.

The above platform contains a DSP to handle communication and image processing, and a CPU (an ARM core) to handle peripheral function blocks and general-purpose processing. The two processors share memory by a system bus (ASB). Many of the following function blocks need these two processors to handle: the Real-Time Clock (RTC) for scheduling; LCD/VGA Controllers for display; Keyboard Controller (K/B), Digitizer and Joystick Controllers for human interface input; Voice Codec for speaker and microphone; Baseband Codec connected to a radio front-end (RF); SPI interface for storage cards, Smartcard Controller for authentication; USB/UART to communicate with other system; and a hardware accelerator

Table 2.1 Popular platforms

Market	Target application	Platform name	Manufacturer
Consumer	Digital Camera	Raptor II	Conexant
	PDA	PXA240	Intel
	PDA	DragonBall	Motorola
	DVD R/W	Dimension8600	LSI
	Set Top Box	Omega Sti5512	ST
	Digital TV	TL850	Teralogic
	Digital Audio	TMS320Daxx	TI
	MP3	Maverick	Cirrus
	DAB	TMS320DRE200	TI
	Home Plug	Piranha	Cogency
	USB device	PSoC	Cypress
	General	CSoC	Triscend
	General	Excalibur	Altera
	General	SoC-Raptor	Wipro
	General	PalmPak	Palmchip
	General	RSP	Philips
	General	PSA	Improv
Wireless	CDMA	MSM3000	Qualcomm
	GSM 2.5G	SGOLD	Infineon
	GSM 2.5G	OMAP710	TI
	3G	I300	Motorola
	BlackBerry	SoftFone+	ADI
	802.11b/Bluetooth	TrueRadio	Mobilian
	802.11a	Tondelayo	Systemonic
	802.11a AP	AR5001AP	Atheros
	Bluetooth	Bluecore	CSR
	GPS	Sifstar	Sirf

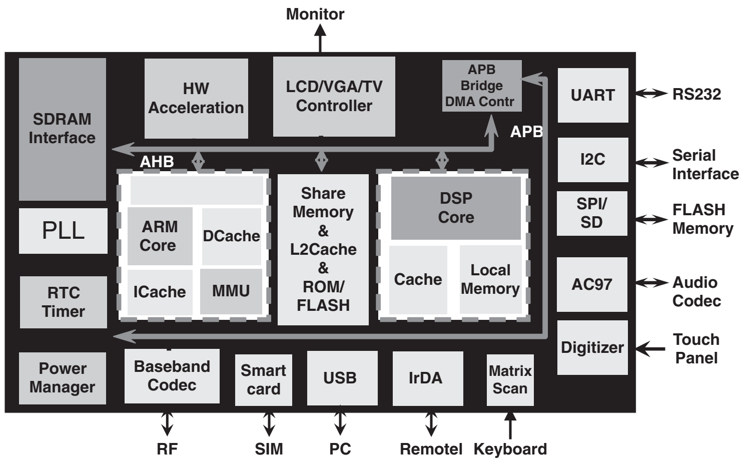


Fig. 2.2 ARM-based wireless communication platform

to assist image processing. All these blocks support real-time functions. All above function blocks on the system are connected by a bus hierarchy. Each function block that is using a bus will affect the performance of other function blocks on the bus. Before access to the bus, each function block needs to get authentication from the bus arbiter. Too many function blocks connected on a bus will make it unable to work while most time are spent on authentication.

On a desktop PC, possible peripheral to integrate is unknown, thus *Plug-and-Play* is required. System bus should be designed to satisfy the highest possible bandwidth, thus it is over-designed for normal work-load. In embedded systems, a Mix-and-Match methodology is used instead. At first a reference platform is selected, which is mostly provided by the processor vendor. Then function blocks are inserted or removed, and workloads are rescheduled by system level analysis. If bandwidth constraint is not satisfied, modify either a function block or the bus architecture and try again. For example, if the new function occupies too many bandwidths to access a shared memory, adding a dedicated memory and a local bus as shown in Fig. 2.3 can solve this problem.

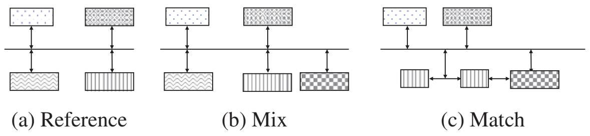


Fig. 2.3 Mix-and-match methodology

2.1.1 OMAP

OMAP (Open Multimedia Application Platform) is a series of dual-core processor developed by TI for multimedia and wireless applications. OMAP1510 is built of an ARM925 processor core, a TI TMS320C55x DSP core, a 192 KB share-memory interface, and some peripherals for multimedia applications [4]. The ARM925 core containing a 16 KB instruction cache and an 8 KB data cache can work up to 175 MHz. The TMS320C55 DSP core containing a 16 KB instruction cache and an 8 KB data cache can work up to 200 MHz. It embeds 64 KB dual-port RAM and 96 KB single-port RAM, and a graphic accelerator with two multiply-accumulator (MAC) units.

Running with the same clock frequency, power consumption of a DSP core is higher than an ARM core. The DSP core is efficient to compute data-oriented codes, and the ARM core is better on control-oriented codes. A typical 1024-point FFT computation run on an ARM needs 1 mega cycles, but only 40 K cycles on a DSP. Both the ARM and DSP software codes can be developed under a single Code Composer Studio (CCS) environment, which helps to reduce the dual-core programming complexity. TI offer abundant of Chip Support Library application interface (CSLAPI) to low level programming effort.

The key technique used in the OMAP software architecture as shown in Fig. 2.4 is the DSP/BIOS Bridge. Both ARM and DSP are working under a single operating

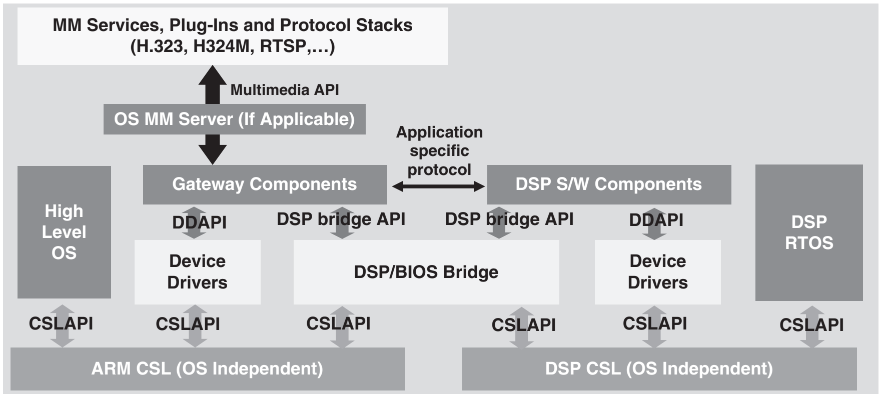


Fig. 2.4 OMAP software architecture

system (OS). Programmer can treat DSP as a device attached under Linux OS directory `/dev`. A high level multimedia API offers a unified interface to designer. High computation operation is dispatched to DSP/BIOS Bridge by DSP API. DSP/BIOS Bridge handles the scheduling and inter-core communication.

ARM and DSP use shared-memory message-passing architecture for communication. Message is packed as a mailbox, including command, data and flag. Both cores can create more than one task, each of which will create a mailbox channel under the `/dev` directory for communication. When a message is ready in shared-memory, an interrupt will wake up the device driver to inform a corresponding task to receive the message.

Following shows the implementation of an MP3 decoder as an example. MP3 (MPEG-1 layer-3) is an audio coding standard. MP3 stream is composed of many frames. A single channel frame contains 2 granules and each granule has 576 16-bit samples. The 576 samples are down-sampled into 32 sub-bands, where each sub-band contains 18 samples. The down-sampling is a 5-iteration process. And in each iteration, the samples are partitioned into two sub-bands respectively by low-pass and high-pass filters. The 32 sub-bands are transformed into frequency domain by Modified Discrete Cosine Transform (MDCT), and quantized by a psychoacoustic model. Finally Huffman Entropy Encoding is applied to reduce bit rate.

Before implementing the decoder on OMAP, we have to analyze which parts of MP3 decoding take most execution time and have to implement on a DSP. A profiling tool, GNU `gprof`, is used to analyze the decoding C code on a desktop PC and Fig. 2.5 shows a partial profiling report. By the profiling report, we know that function IMDCT32 is the bottleneck.

We now compare two implementations on the OMAP platform. The first implementation runs the whole C code by ARM925. The second uses DSP library to implement IMDCT32 function. Figure 2.6 shows their resource utilization report. The upper part gives the first implementation report, showing that its CPU utilization rate is 40.0%. The lower part for the second implementation shows that its

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
 38.64      0.34     0.34    14554    23.36    23.36  IMDCT32
 14.77      0.47     0.13     2266    57.37    57.37  playingwiththread
  6.82      0.53     0.06                             __libc_write
  5.68      0.58     0.05    13104     3.82     6.11  layer3fixtostereo
  3.41      0.61     0.03    14554     2.06     2.06  extractlayer3
```

Fig. 2.5 Profiling report

```
  1:24am  up  1:24,  1 user,  load average: 0.92, 1.67, 1.48
17 processes: 15 sleeping, 2 running, 0 zombie, 0 stopped
CPU states: 40.0% user,  3.7% system,  0.0% nice, 56.1% idle
Mem:   30672K av,  11856K used,  18816K free,      0K shrd,      0K buff
Swap:      0K av,      0K used,      0K free,                 8680K cached
```

```
  1:30am  up  1:30,  1 user,  load average: 2.60, 1.90, 1.46
17 processes: 13 sleeping, 4 running, 0 zombie, 0 stopped
CPU states: 20.1% user, 79.8% system,  0.0% nice,  0.0% idle
Mem:   30672K av,  11784K used,  18888K free,      0K shrd,      0K buff
Swap:      0K av,      0K used,      0K free,                 8680K cached
```

Fig. 2.6 Resource utilization

CPU utilization rate is 20.1%. As shown, using a DSP can share CPU loading, thus improve performance for MP3 player application.

2.2 System Modeling

On platform-based design, the selection of platform is based on designer's experience. It is a fast way to implement a system, but does not guarantee that it is optimized.

At system level, we need to consider more about environment and user friendliness. Does the product need to work at the Sahara Desert or the North Pole? What human interface does a user prefer? If the bit error rate will become too high at some environment, should we add error correction policy? All these questions and solutions should be decided at system level before entering detailed design.

At system level, we need a model to describe function behavior of a system. This level of modeling targets for a unified representation for hardware and software, which contains the following features:

(1) It can describe high level system architecture.
(2) It is independent to the implementation of hardware and software.
(3) It supports refinement for hardware/software partitioning.
(4) It enables architecture exploration for hardware/software cross-fertilization.
(5) It supports co-simulation environment for communication. And
(6) It supports functional co-verification.

Using a unified representation, we can identify system bottleneck, evaluate performance, and calculate the cost of a hardware/software partitioning at early design stage.

For different purposes, many modeling methods have been introduced. They will be briefly introduced in the following subsections.

2.2.1 State-Oriented Models

The system temporal behavior of a machine can be represented as state transfer in a finite state machine. The state number of a real machine is finite. Two models are used to represent a finite state machine:

(1) Mealy machine: The output is determined by current state and input.
(2) Moore machine: The output is determined only by the state.

In circuit view, the input signals of a Moore machine are all buffered in internal state registers, and mealy machine allows input signals to pass to output by combination of states.

Finite state machine behavior is represented by state transfer chart. Figure 2.7 shows a simple example of Mealy machine model. Each node represents a state. Thus a 3-state machine requires 2 or 3 registers to implement the states, depending on timing and cost constraints. The arc between two states represents a state transfer, which is derived by an input signal which is attached on the arc. The output signal is also attached on the arc, which can be a datapath computation.

The state transfer chart is not suitable to represent hierarchy and concurrency. When two state machines are working concurrently, their state registers should be combined together, and the state number will increase exponentially. The size of the state transfer chart becomes soon explosive for a complex system.

Petri-net is a graphical and mathematical tool to provide a uniform environment for modeling, formal analysis, and design of discrete event systems [5]. It was named after Carl A. Petri who created the concept in 1962. Petri-net offers the ability to represent concurrency. Petri-net can be used to model properties, such as process synchronization, asynchronous events, concurrent operations, and conflicts or resource sharing. Petri-net is identified as a particular directed graph by three types of objects: *place*, *transition* and directed *arc*. *Place* represented as a circle is used for an operation. *Transition* is represented as a bar. *Arc* demonstrates state transfer

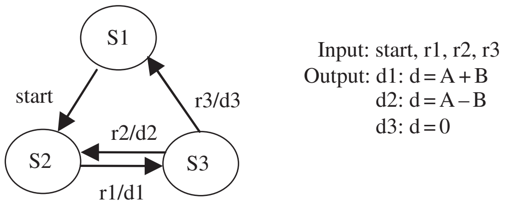
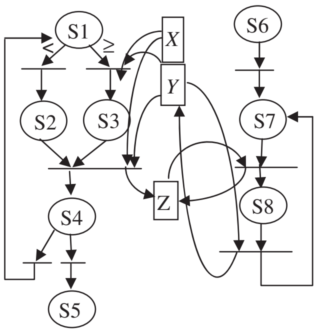


Fig. 2.7 Mealy finite state machine



```
    do {
S1: if (X<Y)
S2:    wait(Z=5);
     else
S3:    wait(Z=0);
S4:} while(Y<100);
S5:end
```

```
S6: Z=0;
     while(1) {
S7:  Z=S2?Z++:
          S3?Z--:Z;
S8:  Y=Y+1;
     }
```

Fig. 2.8 Petri-nets of two concurrent processes

and data node association. Data nodes, represented as rectangles and associated on transition, are used for the event trigger point and data dependency, which combine the control graph and data graph together. Decision branch modeled in Petri-net is a single *place* input to multiple competing *transitions*. Figure 2.8 shows a Petri-net example. It contains two concurrent processes, one has 5 states and another has 3 states. The two processes share variable Y and variable Z, they will affect each other.

2.2.2 Activity-Oriented Models

The *activity-oriented model* focuses on representing what tasks to do and their dependences, but it lacks of temporal information. Flow chart is a widely-used activity-oriented model. It is useful on control-oriented representation. Figure 2.9 shows an example.

Control and Data Flow Graph (CDFG) [6] is a widely-used intermediate representation in compiler. CDFG is the model for capturing design descriptions for compiler and high-level synthesis which work well for traditional scheduling and binding techniques.

CDFG retains high level information about code structures and semantics in terms of control flow graph (CFG) and data flow graph (DFG), which are often used in compilers. Control flow graphs (CFG) are used to handle code sequences. Each node in a CFG represents a basic block. Data flow graphs (DFG) maintain the data dependencies between operations. Control flow graph is transformed from the control structures in a code, such as `if-else` and `for-loop` control statements, for describing the code statement execution orders. Because of the sequential ingredient of most software languages, like C and C++, building a control flow graph from a sequential code is simply a direct mapping. Data flow graph is basically a

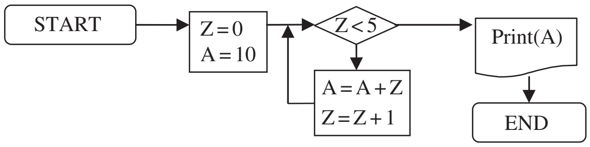
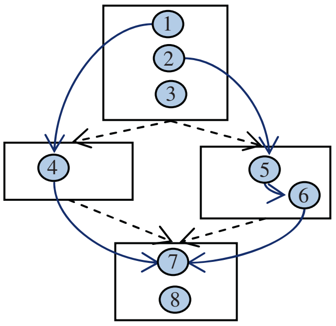


Fig. 2.9 Flow chart

Fig. 2.10 CDFG

```
S1: t=a+b;
S2: u=a-b;
S3: if(a<b)
S4: v=t+c;
else
{
S5: w=u+c;
S6: v=w-d;
}
S7: x=v+e;
S8: y=f-e;
```



data dependence graph which nodes are operations and edges are data dependence relations. Figure 2.10(a) shows a code segment and Fig. 2.10(b) illustrates its control and data flow graphs. The solid rectangles represent basic blocks. A dashed arrow between basic blocks denotes a control flow and a solid arrow denote a data flow between operations.

Hierarchical Task Graph (HTG) [7] is based on CDFG, maintaining the hierarchical structuring of a code such as `if-then-else` blocks, `for` and `while` loops. HTG extends the range of optimizations, especially beneficial to source-to-source optimization and other coarse-grain transformations. HTG also enables higher order manipulation, *e.g.*, coarse-grain code restructuring and operation moving across large pieces of code. A design HTG is constructed by creating a compound node corresponding to each control construct in the design. Figure 2.11 shows the nested `if-else` statements in terms of nested HTG forms.

The following paragraphs list the formal definition of an HTG.

A hierarchical task graph HTG is a hierarchy of directed acyclic graphs G_{HTG} (V_{HTG}, E_{HTG}), where the vertices $V_{HTG} = \{htg_i | i = 1, 2, \ldots, n_{htgs}\}$ can be one of the following three types.

(1) Single nodes represent nodes that have no sub-nodes and are used to encapsulate basic blocks. Basic blocks are a sequential aggregation of operations that have no control flow (branches) between them.
(2) Compound nodes are recursively defined as HTG, that is, they contain other HTG nodes. They are used to represent structures like `if-then-else` blocks, `switch-case` blocks or a series of HTG.
(3) Loop nodes are used to represent the types of loops (`for, while-do, do-while`). Loop nodes consist of a loop head and a loop tail that are single nodes and a loop body that is a compound node.

The edge set E_{HTG} in G_{HTG} represents the flow of control between HTG nodes. An edge (htg_i, htg_j) in E_{HTG}, where $htg_i, htg_j \in V_{HTG}$, signifies that htg_j will be executed after htg_i has finished execution. Each node htg_i in V_{HTG} has two distinguished nodes, $htg_{Start}(i)$ and $htg_{Stop}(i)$, belonging to V_{HTG} such that there exists a path from $htg_{Start}(i)$ to every node in htg_i and a path from every node in htg_i to

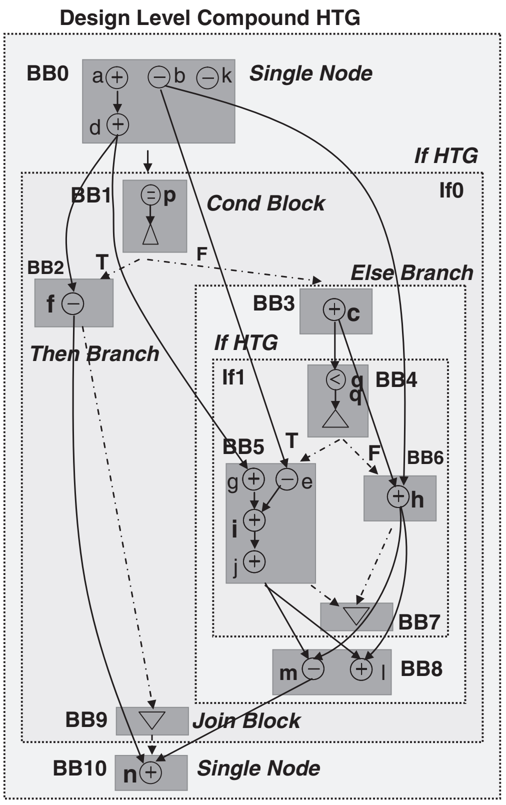


Fig. 2.11 Compound if-else statements in HTG form

htg$_{\text{Stop}}(i)$. The htg$_{\text{Start}}$ and htg$_{\text{Stop}}$ nodes belonged to an HTG compound or loop node are always single nodes.

Actually, when the dotted rectangles in Fig. 2.11 are removed, the HTG diagram becomes a CDFG diagram. The basic blocks are shown by shaded boxes within the HTG nodes (BB0 to BB10) and operations are denoted by circular nodes with an operator symbol inside. Dashed lines denote control flow between HTG nodes. Solid lines denote data flow between operations. A fork in the control flow is denoted by a triangle (Δ) and a merge by an inverted triangle (∇).

Program Dependence Graph (PDG) [8] represents a code in its original design concepts. PDG is another variant of the CDFG which relaxes the sequential flow restrictions. Figure 2.12 demonstrates the characteristics of PDG.

The dotted lines denote the control flow dependences, *e.g.*, the ENTRY node must go to node (2), then node (3). The solid lines denote the data dependences, *e.g.*, node (4) depends on node (1), and node (9) depends on nodes (1) and (4). There are two places worth notice in Fig. 2.12. Since the statements in lines (8) and (9) are counterparts of nodes (8) and (9), it is obvious to see that the entry nodes (3) and (4) respectively have a data dependence relation with nodes (8) and (9) because of the solid lines. The second place is the introduction of relation nodes under node (2) with two edges labeled T (True) and F (False), respectively. Relation nodes are used when a new basic block is encountered, *e.g.*, the nodes (3) and (4) create a

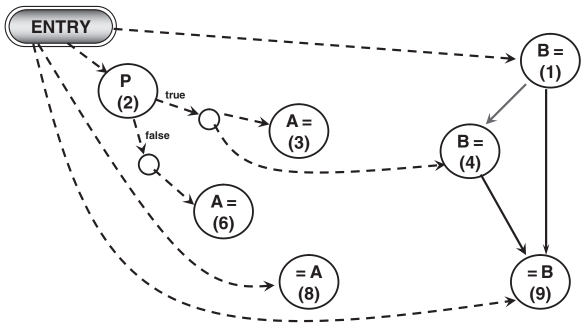


Fig. 2.12 An example of PDG

new basic block. PDGs relieve the ordering property of CDFGs. There is no need to add ordering information in CDFGs. However, PDGs still need various compilation techniques to detect *true data dependences*. For instance, the data dependence edge between nodes (1) and (4) is an *output data dependence*, where nodes (1) and (4) write data to the same memory location, B.

System Dependence Graph (SDG) [9] is an extension of PDG. It is designed to incorporate collections of procedures (with procedure calls) rather than just monolithic program codes. Figure 2.13 depicts an SDG example. The straight medium-bold edges represent the *control dependences*, *e.g.*, the edges between the "ENTER *Main*" node and its three child nodes. The light solid arcs represent *flow dependencies* of the source code, *e.g.*, the arcs from node "i:=1" to node "while i<11". The heavy-bold arcs represent *transitive inter-procedural flow dependencies* (corresponding to subordinate characteristics graph edges), *e.g.*, the arcs from node "$\delta_A(x)$:= sum" to node "sum := $\delta'_A(x)$". Dashed arrows represent call edges, linkage-entry edges, and linkage-exit edges.

2.3 Video Coding

H.264 advanced video coding (AVC) [10] is one of the latest international video coding standards. This standard is developed by the ITU-T Video Coding Experts Group and the ISO/IEC Moving Picture Experts Group. It can achieve higher coding efficiency than previous standards. The advantages of this new video coding standard H.264/AVC are its higher quality, less storage, less bandwidth, more robust transmission, easier random access, more kinds of manipulations and intelligent processing.

H.264/AVC is developed for the next generation application. It aims at not only video content compression but also video conferences, television broadcasting, and internet streaming. It is designed to replace all the past video standards in almost all kinds of applications. So, it defines different profiles to meet the various requirements in different applications. *Baseline Profile* is used in real-time communication applications and conversational services which are important for low latency and low bit stream size requirements. *Main Profile* is designed for the highest coding-efficient storage to use in entertainment video applications, such as satellite broad-

```
program Main
 sum:=0;
 i:=1;
 while(i<11) do
 call A(sum,i);
 enddo
end
```

```
Procedure
A(x,y)
  call Add(x,y);
  call Inc(y);
return;
```

```
procedure
Add(a,b)
  a:=a+b
return;
```

```
procedure
Inc(z)
  call Add(z,1);
return;
```

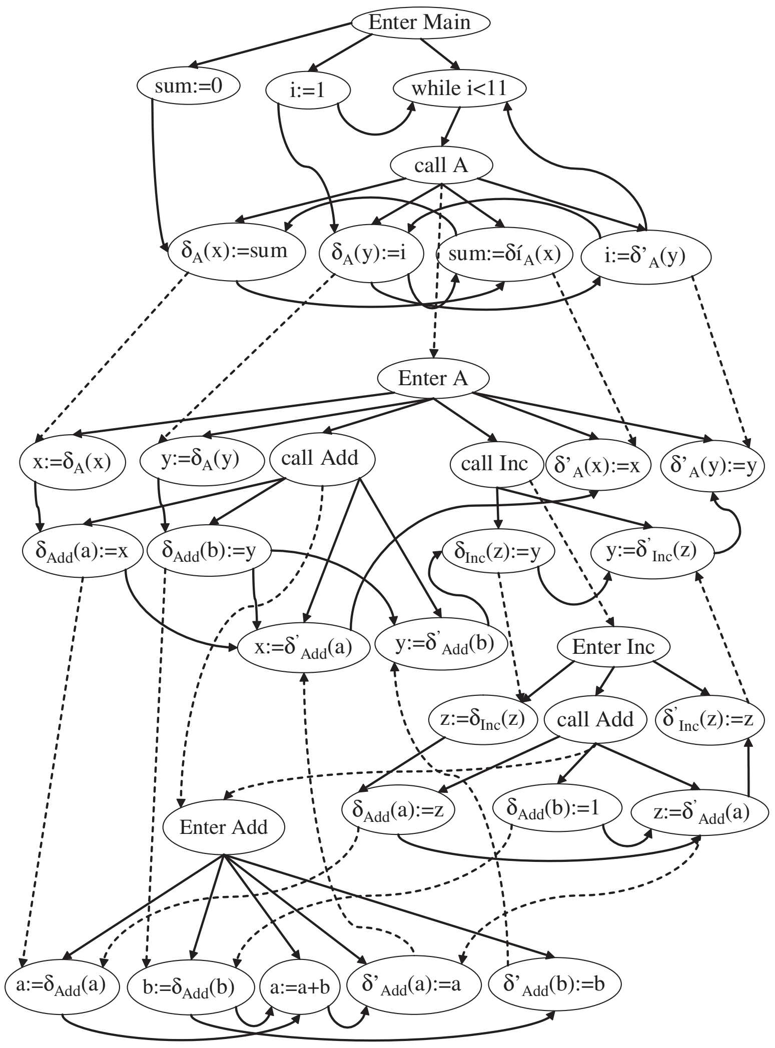


Fig. 2.13 An example of SDG

casting, cable modem, and DVD players for standard and high definition video, *etc*. *Extended Profile* is defined for supporting services that operate at 50–1500 Kbps and have two seconds or more latency.

A picture in an H.264 video sequence could be partitioned into slices. The minimum number of slices in a picture is one. Slice is a collection of macroblocks which are processed in the order of a raster scan. The luminance and chrominance format defined in H.264 is YUV 4:2:2. So, a macroblock has one 16×16 luma (Y) sample and two 8×8 chroma (Cb, Cr) samples. A macroblock is the basic processing unit in H.264 coding process.

The video sequences in H.264 comprise 4 types of frames: I (Intra), P (Prediction), B (Bi-directional), and S (Switched) frames. *I frame* is a basic component in all H.264 profiles that uses intra coding to remove spatial redundancy. *P frame* is also a basic frame in all profiles, but it uses inter prediction method to predict its values from previous encoded P and I frames to remove temporal redundancy. *B frame* is used in Main profile and Extended profile. It uses both forward and backward motion compensations as inter prediction to achieve higher compression rate than other frames. *S frame* is a new frame type introduced by H.264 only and used in an Extended profile. This new frame is used for efficient switching between two different bitstreams.

2.3.1 H.264 Coding Process

A typical H.264 *coding process* is shown in Fig. 2.14. When encoding a picture, we first split the picture into 16×16 macroblocks (MBs). Each of the 16×16 MBs is a basic process unit in the encoding scheme. Once a MB is encoded, the encoder will predict what the MB should be. The MB data is divided by a prediction value. Then the residual data is processed by transform (T) and quantization (Q). After quantization, the data can be encoded by an entropy encoder to compress the data size and sent to the decoder. To be consistent with the decoder, prediction is done by referring to a reconstructed picture, not the original picture. The reconstruction is built from the quantization data by inverse quantization and inverse transformation, and adding the prediction value to recover the picture. Based on this reconstructed picture, the encoder can further do prediction for the next MB from performing intra prediction for the MB in the same frame, or motion estimation from the past decoded pictures. Thus the encoding process can be completed in such a loop process.

The *decoding process* is similar to the reconstruction process, but adding an entropy decoding stage before inverse quantization.

2.3.2 Motion Estimation

Motion Estimation finds the best matching candidate block between a current macroblock and its reference frames in a search window. It can efficiently reduce the

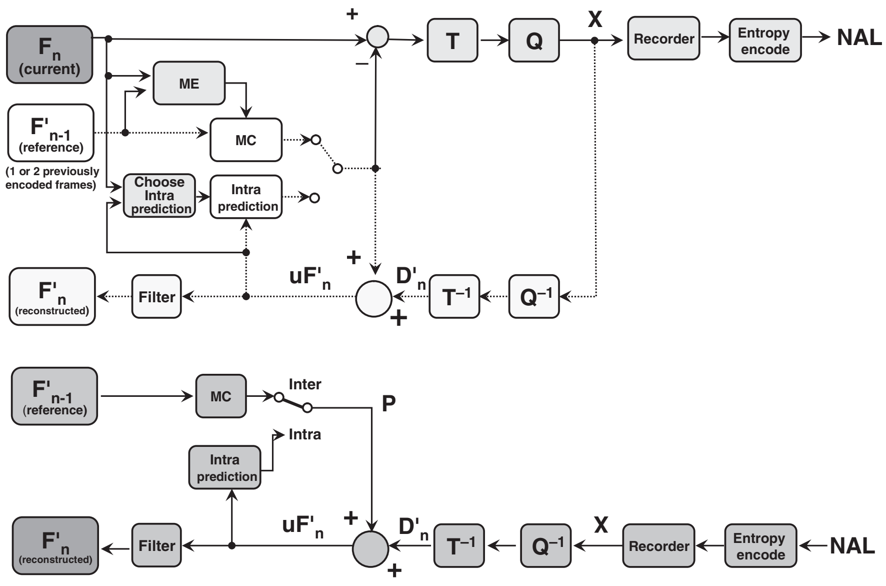


Fig. 2.14 H.264 coding process

temporal redundancies. With motion estimation, we can promote the bit rate effectively by transferring the motion vectors of a MB. Thus, motion estimation is one of the most important functions in many video coding standards. In H.264/AVC, the standard supports multiple reference frames in order to find the objects which appear suddenly. Besides, it also supports variable block sizes coding in matching blocks. Although these two functions indeed increase the coding gain, both of them will greatly increase the computational complexity with the comparison of previous standards which only support one or two reference frames and only one fixed MB size.

The selected frames for prediction are indicated by a reference index which is associated with the frame index in the buffer. The process of encoding which is based on the values predicted to form the best matching candidate block is called *motion compensation*. When the best matching candidate block is found, it will be used to form a reference frame to be pasted on the current frame. Then, a vector called *motion vector* can be drawn from the reference frame.

If an encoder wants to transfer a block data in Frame *t*, there are two ways to get it. (1) Let the encoder transfer all pixels of this block to decoder. (2) Since there is not much difference between Frame *t* and Frame *t*-1, and the data of Frame *t*-1 had already reconstructed by a decoder and is stored in the buffer, we can take Frame *t*-1 as a reference and exploit the data of Frame *t*-1 in the decoder buffer. Thus, we just need some information, the motion vector, to construct the Frame *t*.

There are many criteria used to judge the best matching block. One of the most widely used criteria is *sum of absolute differences* (SAD). The function of SAD is shown as:

$$SAD(m,n) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} |C(i,j) - R(i+m, j+n)|,$$

where (*m*, *n*) is the distance of the candidate block corresponding to coding block C at position (*i*, *j*), and *N* is the macroblock size. H.264/AVC defines the maximum of (*m*, *n*) as *search range* for different levels. The *motion vector MV* is defined as:

$$MV = \arg \min_{(m,n) \in \text{search range}} SAD(m,n).$$

H.264/AVC supports seven block size modes (16×16, 16×8, 8×16, 8×8, 8×4, 4×8, and 4×4). And H.264/AVC allows 10 reference frames in each single direction (forward/backward). This property is different from the prior standard like MEPG-2, which used only one previous picture to compensate.

A 16×16 macroblock is a basic process unit in a coding loop. In the inter prediction, we will test if a macroblock is suitable to be predicted in 16×16, 16×8, 8×16, or 8×8 size. If the most suitable size is 8×8, it can be further segmented into 8×4, 4×8, or 4×4 blocks for motion estimation. By the variable block sizes, we must compute up to 41 modes of motion vectors, as shown in Fig. 2.15.

Most prior standards allow half-sample motion vector accuracy at most. H.264 improves on this by adding quarter-sample motion vector accuracy. With this precise accuracy, the quality will be improved effectively. After integer pixel motion estimation is done. The neighbor pixels will be used to interpolate the half pixels and quarter pixels values, and then do the same ME operations to find out the best matching motion vector.

The motion vector mentioned above is defined for luma motion vector. And the motion vector of the chroma parts can be derived from the luma one. Since the size of a luma vector resolution is twice as large as the chroma vector, the chroma vector can be obtained by dividing the corresponding luma vector by two. Since the accuracy of the luma motion vectors is 1/4 sample pixel, and the chroma is half resolution of luma, thus the chroma motion vector accuracy is 1/8 sample pixel.

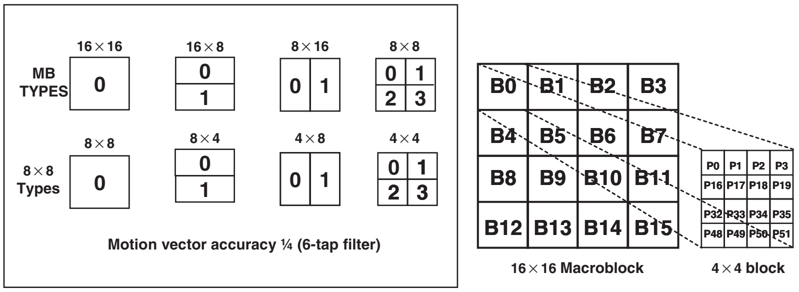


Fig. 2.15 Variable block size and segmented macroblock

To get best MV, we should evaluate all distances (m, n) and 41 modes. The computation effort is very high. Many speedup implementations for motion estimation had been introduced.

In hardware implementation, the goal is to compute 41 motion vectors concurrently. Figure 2.16 is an 1-D implementation example [11]. Another 2-D systolic array implementation can be found in [12, 13]. In the 1-D architecture, the current data are fed to each processing element (PE) by 16 registers, and the reference data 0 and data 1 are separately fed to each PE. Thus, the SAD value of 16 4×4 blocks can be computed concurrently by the 16 PEs. The reference data which motion vector (0,0) needs are not the same with the motion vector (15,0). Thus, the reference data input to each PE needs to be well scheduled. It is implemented using two SRAMs with two broadcasting buses.

Figure 2.17 shows the schedule of data input for each PE, where C represents the current data, R represents the reference data, and vectors (0, 0) to (31, 15) represent their corresponding positions. For every PE, C(0, 0) to C(15, 15) are needed to compare the difference of every pixel in a 16×16 current macroblock. On the contrast, the reference data of different motion vectors are not the same. For PE00, the required reference data are R(0, 0) to R(15, 15) to calculate motion vector (0, 0). For PE01, the required reference data are R(1, 0) to R(16, 15) to calculate motion vector (1, 0). We can see that many of the required reference data are the same when motion vectors (0, 0) and (1, 0) are calculated, such that we can first feed R(0, 0) to R(15, 0) in Ref data 0 at the first 16 clocks. When R(0, 1) in Ref data 0 is required by PE00 at the 17th clock, R(16, 0) is fed to Ref data 1 which is required by other PEs at the same time. In such schedule, we can compute 16 motion vectors concurrently. Every PE completely receives data in successive cycles. If the search range is [−8, +7], we can set the PE00 to compute (−8, −8) to (−8, +7), PE01 to

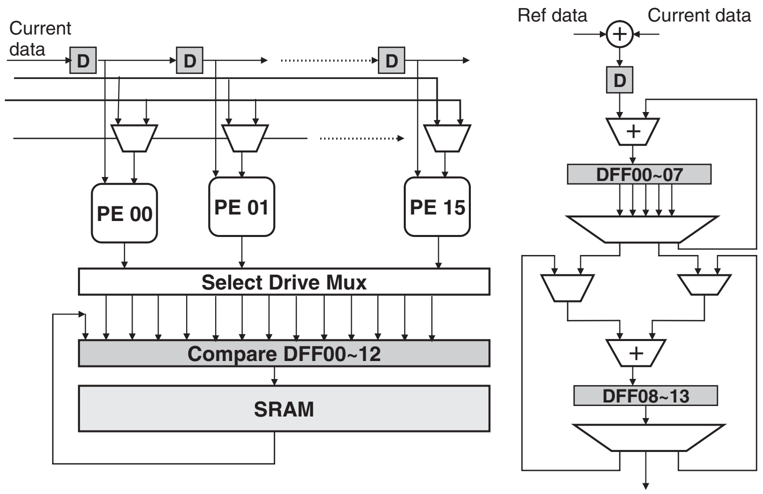
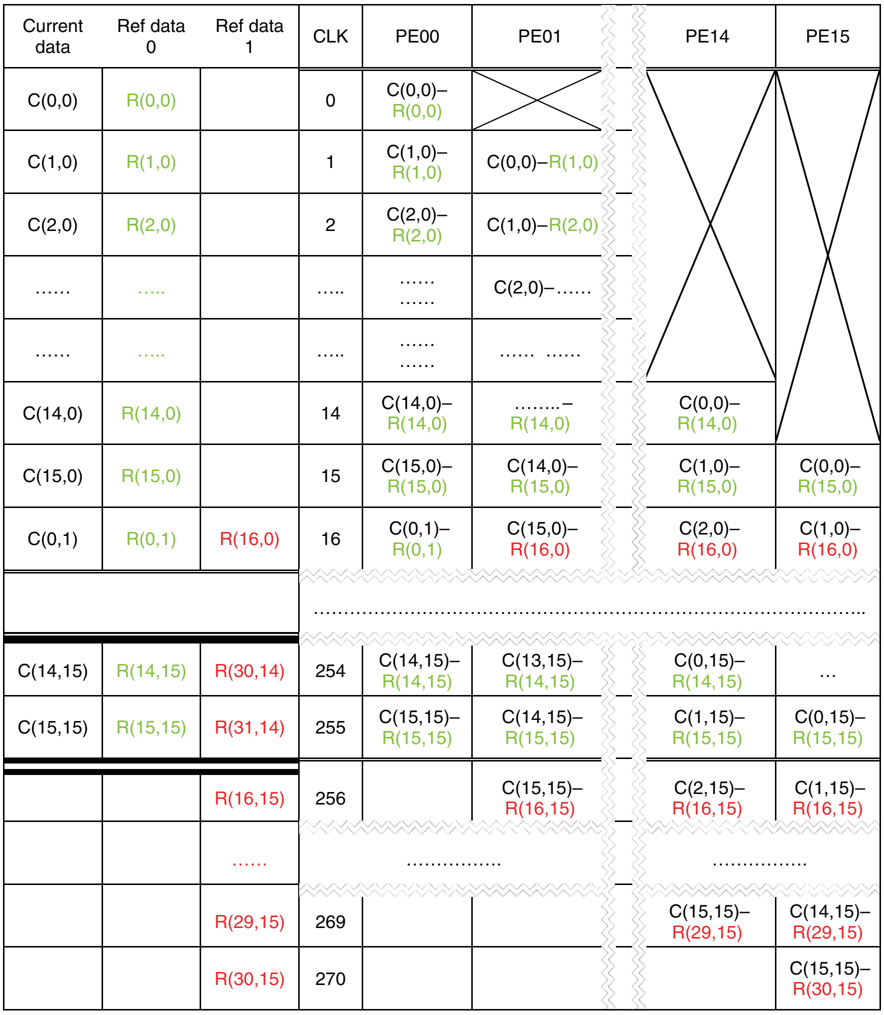


Fig. 2.16 1-D motion estimation architecture



Current data	Ref data 0	Ref data 1	CLK	PE00	PE01	PE14	PE15
C(0,0)	R(0,0)		0	C(0,0)–R(0,0)			
C(1,0)	R(1,0)		1	C(1,0)–R(1,0)	C(0,0)–R(1,0)		
C(2,0)	R(2,0)		2	C(2,0)–R(2,0)	C(1,0)–R(2,0)		
……	…..		…..	…… ……	C(2,0)–……		
……	…..		…..	…… ……	…… ……		
C(14,0)	R(14,0)		14	C(14,0)–R(14,0)	……..–R(14,0)	C(0,0)–R(14,0)	
C(15,0)	R(15,0)		15	C(15,0)–R(15,0)	C(14,0)–R(15,0)	C(1,0)–R(15,0)	C(0,0)–R(15,0)
C(0,1)	R(0,1)	R(16,0)	16	C(0,1)–R(0,1)	C(15,0)–R(16,0)	C(2,0)–R(16,0)	C(1,0)–R(16,0)
			…………………………				
C(14,15)	R(14,15)	R(30,14)	254	C(14,15)–R(14,15)	C(13,15)–R(14,15)	C(0,15)–R(14,15)	…
C(15,15)	R(15,15)	R(31,14)	255	C(15,15)–R(15,15)	C(14,15)–R(15,15)	C(1,15)–R(15,15)	C(0,15)–R(15,15)
		R(16,15)	256		C(15,15)–R(16,15)	C(2,15)–R(16,15)	C(1,15)–R(16,15)
		……		…………….		…………….	
		R(29,15)	269			C(15,15)–R(29,15)	C(14,15)–R(29,15)
		R(30,15)	270				C(15,15)–R(30,15)

Fig. 2.17 Schedule of data input for each PE

compute (−7, −8) to (−7, +7), PE02 to compute (−6, −8) to (−6, +7), and so on, and finally PE15 to compute (+7, −8) to (+7, +7). The motion vectors, which are computed by the same PE, are just different in Vy but the same in Vx. After the SAD of a specific motion vector is computed individually by each PE, the outputs of PEs should be compared for selecting the minimum one as the result of motion estimation. In this architecture, the comparison is implemented by 13 DFFs. The SAD values of larger block size can be composed of the 16 4×4 blocks.

In software implementation, the purpose is to reduce search point. That will get a sub-optimal solution. It is a trade-off between computation time and compression rate. Many algorithms had been introduced, including three-step search, diamond search, modified spiral search, *etc*.

Figure 2.18 shows the concept of a *three-step search.* For a search range of ± 6, the first step searches nine (m, n) points, for $m, n = \{0, 4, -4\}$, then selects the point (m_1, n_1) which SAD is minimum. The second step searches the other nine $(m_1 + a, n_1 + b)$ points, for $a, b = \{0, 2, -2\}$ and selects the point (m_2, n_2) which SAD is minimum. The third step searches the nine $(m_2 + c, n_2 + d)$ points, for c, $d = \{0, 1, -1\}$, and the point (m_3, n_3) which SAD is minimum will be the solution. Compared to the total 169 points, 85% of the computation time is saved.

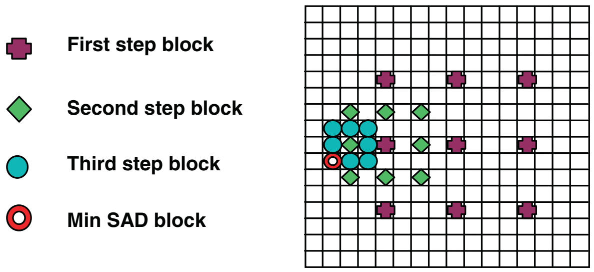


Fig. 2.18 Three-step search

2.3.3 *Intra Prediction*

Intra prediction is used to reduce spatial redundancies. H.264 intra coding uses values of neighbor blocks in a current picture frame to encode a coding block. It is the major reason that makes H.264 intra coding have the highest coding efficiency than any other existing video coding standards. There are two intra prediction types for luminance samples, INTRA_4×4 and INTRA_16×16, and one intra prediction type defined for chrominance samples, INTRA_CHROMA.

In the INTRA_4×4 prediction type, nine prediction modes are defined. One of them is the DC mode representing the mean of neighboring pixels. Others describe the eight possible prediction directions. Each 4×4 block of a luminance sample in a MB can choose one of the nine modes. In Fig. 2.19, the luminance samples labeled as *a* to *p* in the prediction block are calculated based on the neighboring samples labeled as *A* to *L* and *Q*. The neighboring samples come from previously coded blocks. And they will be partitioned into four subgroups. Upper-side neighboring samples include *A*, *B*, *C* and *D*. Left-side samples include *I*, *J*, *K* and *L*. Upper-left side has only a *Q* sample. Upper-right side samples include *E*, *F*, *G* and *H*. If some of these neighboring samples are not available, the prediction mode using these samples will be skipped.

An illustration of nine prediction modes in INTRA_4×4 is presented in Fig. 2.20. Mode 0 is predicted from upper neighboring samples. Mode 1 is predicted from left boundary samples. Modes 3 and 7 will be calculated when the upper and upper-right

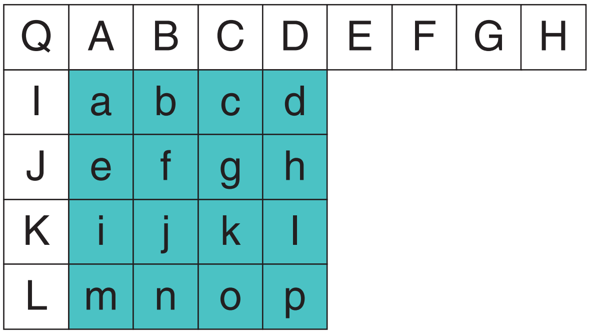


Fig. 2.19 INTRA_4×4 prediction neighbors

neighboring samples are available. Modes 4, 5 and 6 are interpolated from the upper, upper-left, and left neighboring samples. Mode 8 is a candidate of INTRA_4×4 prediction mode when the left-side neighboring samples are available. DC prediction mode takes the average value of the available neighboring samples on the upper and left sides. If both the upper and left sides neighboring samples are not available, DC prediction will be given a prediction value of 128.

An illustration of four prediction modes in INTRA_16×16 is presented in Fig. 2.21. Mode 0 in INTRA_16×16 is used for vertical prediction. Mode 1 is for

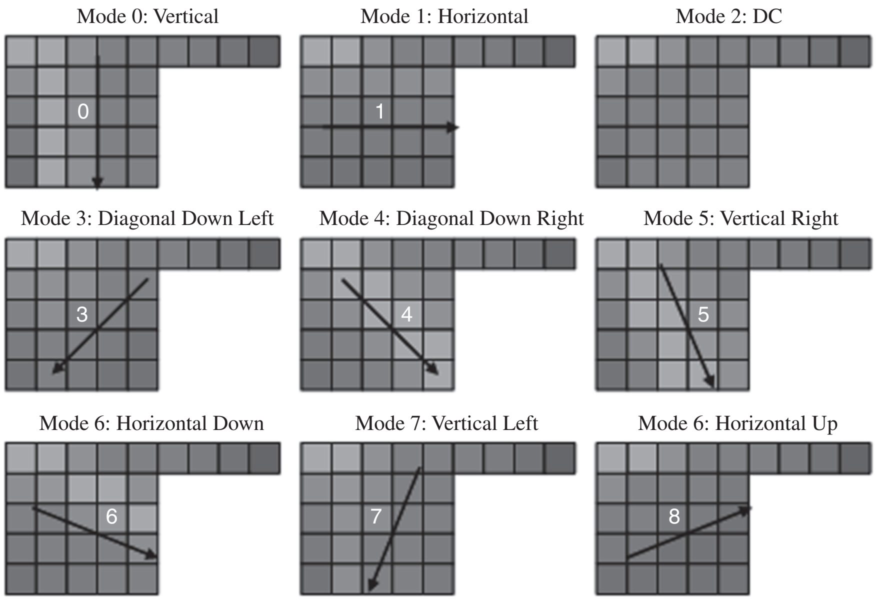


Fig. 2.20 INTRA_4×4 prediction modes

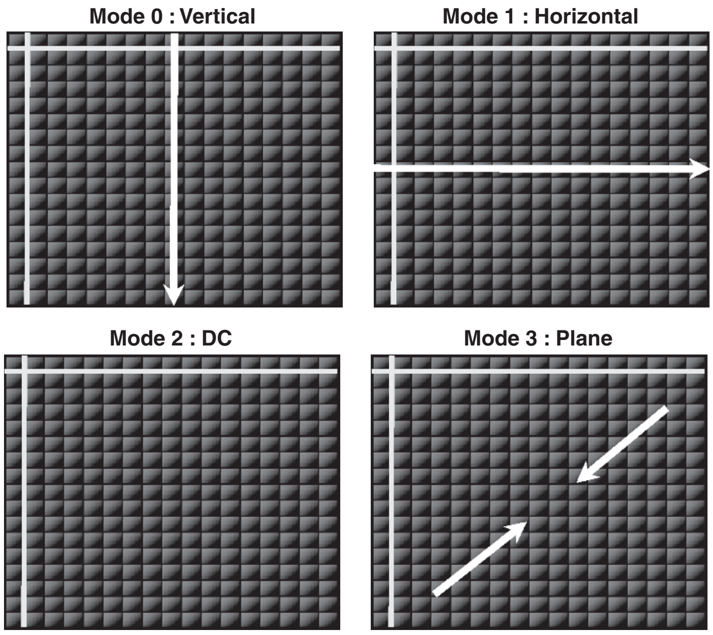


Fig. 2.21 INTRA_16×16 prediction modes

horizontal prediction. Mode 2 is for DC prediction. And Mode 3 predicts by the average value of H and V neighboring samples.

The two chrominance components, Cb and Cr, use the same prediction mode. The INTRA_CHROMA prediction type is like INTRA_16×16, but with a block size of 8×8.

2.3.4 Transform and Quantization

In H.264, *transform* and *quantization* operations are designed for low hardware implementation cost. Shift, addition and subtraction are major operations adopted in H.264. And multiplication operations only appear in quantization and de-quantization stages. The division operation with high hardware implementation cost is avoided in this standard.

H.264 adopts a DCT-like integer transform instead of standards DCT transform commonly used in previously approved video coding standards. The proposed transform has an inverse transform fully defined in integer arithmetic and do not have data drift problem caused by floating-point arithmetic and rounding process. The formula of *integer forward transform* can be written as following equation where $\otimes$ denotes element-by-element multiplication.

$$Y = \left(CXC^T\right) \otimes E_{forw}$$
$$= \begin{bmatrix} 1 & 1 & 1 & 1 \\ 2 & 1 & -1 & -2 \\ 1 & -1 & -1 & 1 \\ 1 & -2 & 2 & -1 \end{bmatrix} \begin{bmatrix} x_{00} & x_{01} & x_{02} & x_{03} \\ x_{10} & x_{11} & x_{12} & x_{13} \\ x_{20} & x_{21} & x_{22} & x_{23} \\ x_{30} & x_{31} & x_{32} & x_{33} \end{bmatrix} \begin{bmatrix} 1 & 2 & 1 & 1 \\ 1 & 1 & -1 & -2 \\ 1 & -1 & -1 & 2 \\ 1 & -2 & 1 & -1 \end{bmatrix}$$
$$\otimes \begin{bmatrix} a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \\ a^2 & \frac{ab}{2} & a^2 & \frac{ab}{2} \\ \frac{ab}{2} & \frac{b^2}{4} & \frac{ab}{2} & \frac{b^2}{4} \end{bmatrix}$$

This equation could be divided into two parts. The first part is a butterfly-based matrix multiplication in which matrix C contains coefficients of ± 1 and ± 2 only. A multiplication by ± 2 can be implemented as a left shift and an addition/subtraction. So, the matrix multiplication does not really need a multiplier. The second part operation is to scale the result of the first part operation. There are only three scaling factors in E_{forw} which can be absorbed by quantization operation. So the final definition of forward transform in H.264 is only the first part of this equation.

The *inverse transform* is described in the following equation. The scaling matrix E_{inv} can also be merged into the de-quantization operation. The final proposed inverse transform is $X' = C_i Y' C_i^T$ and the operations in inverse transform need only right shift and addition/subtraction.

$$X' = C_i(Y'' \otimes E_{inv})C_i^T$$
$$= \begin{bmatrix} 1 & 1 & 1 & \frac{1}{2} \\ 1 & \frac{1}{2} & -1 & -1 \\ 1 & -\frac{1}{2} & -1 & 1 \\ 1 & -1 & 1 & -\frac{1}{2} \end{bmatrix} \left(\begin{bmatrix} y''_{00} & y''_{01} & y''_{02} & y''_{03} \\ y''_{10} & y''_{11} & y''_{12} & y''_{13} \\ y''_{00} & y''_{00} & y''_{00} & y''_{00} \\ y''_{00} & y''_{00} & y''_{00} & y''_{00} \end{bmatrix} \otimes \begin{bmatrix} a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \\ a^2 & ab & a^2 & ab \\ ab & b^2 & ab & b^2 \end{bmatrix} \right)$$
$$\times \begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & \frac{1}{2} & -\frac{1}{2} & -1 \\ 1 & -1 & -1 & 1 \\ \frac{1}{2} & -1 & 1 & -\frac{1}{2} \end{bmatrix}$$

Table 2.2 Quantization table

QP	0	1	2	3	4	5	6	7	8
Qstep	0.625	0.6875	0.8125	0.875	1	1.125	1.25	1.375	1.625
QP	11	12	18	24	30	36	42	48	51
Qstep	2.25	2.5	5	10	20	40	80	160	224

Quantization operation in H.264 uses scaling quantization step sizes. The quantization step is specified by assigning a quantization parameter, *QP*. Table 2.2 lists the relationship between *QP* and quantization step Qstep.

The quantization step will double when *QP* is increased by six. The quantization combines the quantization steps and scaling factors in E_{forw}, it can be divided into three groups related to the scaling factors in E_{forw}. indexed by position indexes *i* and *j*, and each group has only 6 possible values.

In INTRA_16×16 and INTRA_CHROMA prediction types of H.264, a hierarchical coding scheme has to be adopted. DC coefficients will be further compressed using a second stage transform. Residual data of coding blocks will be processed by a DCT-like transform. Then groups DC coefficients of each DCT-like are transformed into a new block, and the second stage transform and quantization applied on it. After reconstructing DC coefficients by de-quantization and inverse transformation, reconstructed DC coefficients and AC coefficients will be combined and input into an inverse DCT-like transform.

In a region with smooth variation texture, INTRA_16×16 will be taken to achieve a higher compression ratio. The DC coefficients in a DCT-like transform of residual data still have a significant correlation between sixteen 4×4 blocks in a MB. H.264 groups the DC coefficients into a 4×4 block, and applies the second stage transform on it to further improve compression efficiency. A 4×4 Hadamard transform is selected. Since Hadamard transform uses orthogonal matrix, the inverse transform has the same formula as the forward transform.

2.3.5 De-Blocking Filter

Due to the smallest encoding is a 4×4 block, the decoded macroblock may have some distortion with neighbor blocks. A *deblocking filter* is applied to smooth the edge difference and improve the appearance of the images. Filtering is applied to 8 edges of a luma 16×16 macroblock and 4 edges of a chroma macroblock as shown in Fig. 2.22.

For the boundary which is filtered, the pixels in both directions which are vertical to the boundary will be modified. The filtering rule is defined by the quantization parameter *QP*, coding mode, and pixel difference across the boundary. With different *QPs*, the thresholds used to adjust are different. According to the coding mode which is inter or intra, and to the difference across the filtering bound.

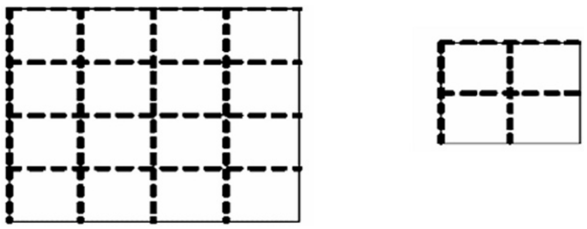


Fig. 2.22 Filtered edges of 16×16 luma and 8×8 chroma macroblock

2.3.6 Entropy Encoding

H.264 provides two entropy coding modes: *Context-based Adaptive Binary Arithmetic Coding* (CABAC) and *Context-based Adaptive Variable-Length Coding* (CAVLC). If the coding mode is CABAC, encoder first converts the coefficients into binary symbols and then performs arithmetic coding to compress the bitstream. For a 4×4 DCT block, the coefficients are first scanned in zigzag order and analyzed whether they are *Significant* or *Last*. *Significant* means that the coefficient is non-zero and *Last* means if the coefficient is the last non-zero coefficient. Only the coefficients before the last non-zero coefficients need to be encoded. The non-zero coefficients are further analyzed for their level and sign. If the level is too high, the coefficient will coded by Exp-Golomb code. If the coding mode is CAVLC, the number of coefficients and trailing ones (T1s) are first coded. T1s ranges from 0 to 3. Except T1s, the other coefficients are coded as normal. After coding the T1s, the sign of T1s are encoded in reverse order. Then it turns to code the normal coefficients. Finally, the run zeros number and run before are coded in reverse order.

2.4 Image Processing

Cameras have become popular in our lives. Transferring photos by a 3G phone is our daily work. *Gesture recognition* is hot in entertainment machines such as the Wii, and even is more useful for healthcare. For example, using a camera to detect a coming car or stairs will help elders and the blind walk safely.

The input from a single or multiple cameras to a computer vision system forms video streams. The system analyzes the video content by separating the foreground from the background, detecting and tracking the objects, and performing some analysis. The analysis results make the scenario in interest more clear such that the operator can easily process it.

Computer vision is a compute-intensive application. A complete computer vision that can recognize many faces and trace objects needs heavy computation which requires high performance processor. Only simpler application such as detecting a moving object is affordable by portable devices. Commonly used computer vision algorithms are filtering, feature extraction, probability-based tracking, and motion analysis [14].

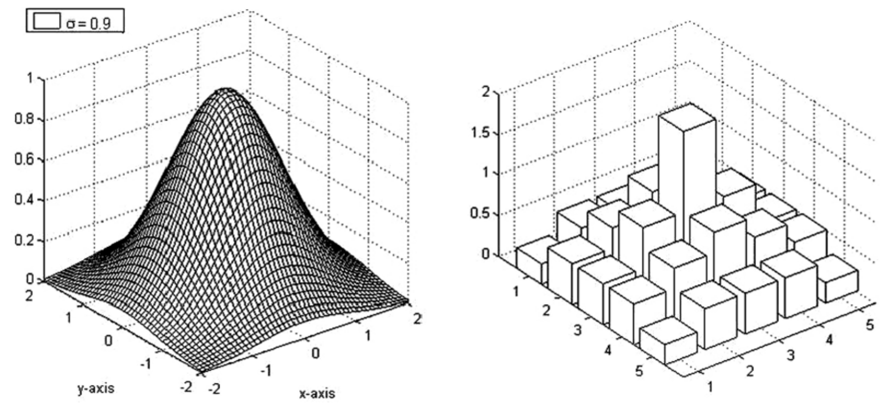


Fig. 2.23 Filters with Gaussian and Semi-Gaussian coefficients

A picture may be corrupted by noise. The "salt and pepper" noise often occurs when environment is dusty. Wiener filter or Median filter should be applied before processing a picture image, and the filter selection will depend on the noise type. A *Gaussian filter* is a low-pass filter that averages neighbor pixels by using the coefficients of a Gaussian distribution equation as shown in Fig. 2.23(a). The Gaussian distribution equation is listed as follows, and it requires floating point computation.

$$G(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}$$

A *semi-Gaussian smooth filter* [15] is typically used for fixed-point applications. As shown in Fig. 2.23(b), it is the quantized version of a Gaussian filter that needs shift and multiplication operations only.

Contract stretching filter improves object contract. It applies a non-linear transform on each pixel such that the low luminance pixels become lower and high luminance pixels become higher. *Sharpness filter* extracts object border by a high-pass filter, and adds the filtered image onto the original image to emphasis object border. The two filters help further analysis easier.

Image features including edges, color regions, textures, and contours. *Edge detection* is the fundamental step in computer vision to extract an object from an image. An abrupt change in a color intensive image occurs at an ideal steep edge, as shown in Fig. 2.24(a). However, in most of the practical applications, it is not the case, due to several factors such as the nature of the scene, reflectance, noise, and blurring. Thus, an edge is thought of having a nonlinear change that could be abrupt as well as slow in the gray level, as shown in Fig. 2.24(b). Therefore, some pixels in the image can be classified as edge pixels by measuring their strength. Here, we assume that the change is measured from the center of pixel $P(i)$ to the center of pixel $P(i-k)$. The edge height, h_e, is defined as the absolute difference between the gray levels of $P(i)$ and $P(i-k)$. The width of an edge W_e is defined as the number of pixels through which the change takes place.

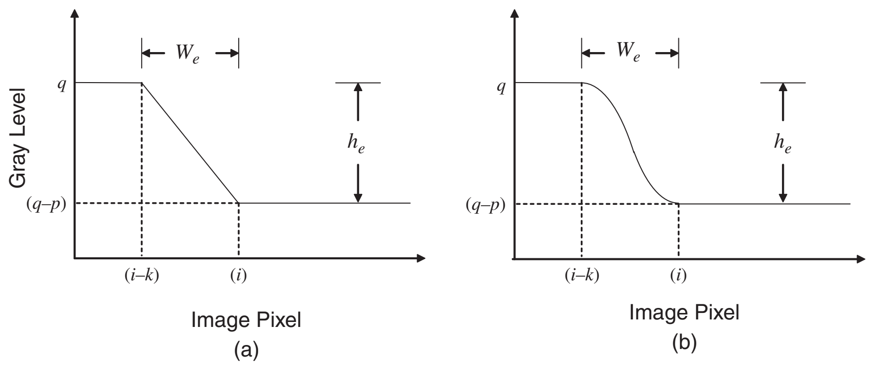


Fig. 2.24 Linear edge and non-linear edge

Absolute Difference Mask (ADM) edge detector is one efficient method to extract edges from an image [16]. It can extract an image edge and record its direction at the same time. The mask is shown in Fig. 2.25. The mask is centered at the pixel of interest, $P(i, j)$, to determine its edge strength. The edge strength of $P(i, j)$ is measured in four directions, and the largest strength value will be assigned to the pixel. These four directions are: the *Nd* negative diagonal direction assigned with a direction value of 1, the *V* vertical direction assigned with a direction value of 2, the *Pd* positive diagonal direction assigned with a direction value of 3, and the *H* horizontal direction assigned with a direction value of 4. Both edge strength and direction can thus be found in parallel.

Once the edge direction is found at pixel $P(i, j)$, it will be recorded with the edge strength of that pixel. The edge strength and direction information will be used in the ADM edge detection and localization processes. The steps to find both the

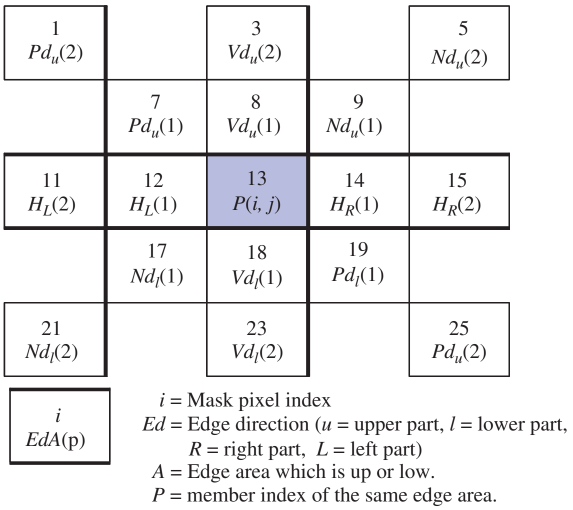


Fig. 2.25 Absolute difference mask

edge strength and the *edge direction* are shown as follows. The direction of an edge having the smallest absolute difference is chosen as the edge direction, dir_e.

(1) Prepare inputs to find the absolute differences for $P(i, j)$:

$$\begin{aligned} V_u &= V_u(1) + V_u(2), & V_l &= V_l(1) + V_l(2), \\ H_R &= H_L(1) + H_R(2), & H_L &= H_L(1) + H_L(2), \\ Pd_u &= Pd_u(1) + Pd_u(2), & Pd_l &= Pd_l(1) + Pd_l(2), \\ Nd_u &= Nd_u(1) + Nd_u(2), & Nd_l &= Nd_l(1) + Nd_l(2), \end{aligned}$$

where V_u: Vertical edge upper part, V_l: Vertical edge lower part, H_R: Horizontal edge right part, H_L: Horizontal edge left part, Pd_u: Positive diagonal edge upper part, Pd_l: Positive diagonal edge lower part, Nd_u: Negative diagonal edge upper part, and Nd_l: Negative diagonal edge lower part.

(2) Calculate all absolute differences for $P(i, j)$:

$$\begin{aligned} V &= |V_u - V_l|, & H &= |H_R - H_L|, \\ Pd &= |Pd_u - Pd_l|, & Nd &= |Nd_u - Nd_l|. \end{aligned}$$

(3) Determine edge strength and direction: (S_e: Edge Strength, dir_e: Edge direction):

$$\begin{aligned} S_e &= max\{V, H, Pd, Nd\}/2, \\ dir_e &= dir(min\{V, H, Pd, Nd\}). \end{aligned}$$

Most application needs to locate an object, especially an incoming vehicle. There are some methods specified to detect incoming vehicles: using symmetry features captured by single camera [17], using the shadow, entropy, and symmetry features captured by single camera [18], using evolutionary Gabor filter optimization for vehicle detection [19], and using rectangular patch and tracking to detect vehicle [20]. Symmetry is a simple but efficient way, while most interesting objects are rectangle or with some symmetric. Figure 2.26 is an example. The right picture is

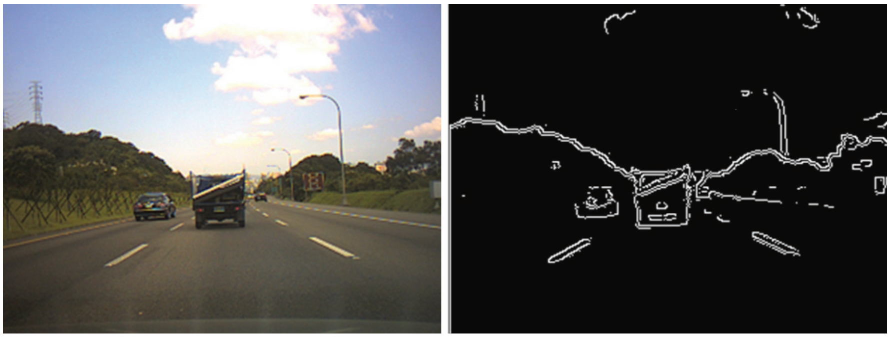


Fig. 2.26 Symmetric feature

the horizontal edges of left picture. It is easy to see there are two sets of horizontal parallel lines that represent the two cars.

In order to track an object, knowledge about what the object looks like is needed. Such knowledge is described by the statistical distribution of the region of interest. *Estimation-Maximization* (EM) algorithm is used to train model parameter. Assume there are M objects which can be modeled as:

$$y = f_m(x) = a_{m,0} + a_{m,1}x + a_{m,2}x^2 + \cdots, \quad 1 \le m \le M$$

Then we have to determine which pixel belong to which object, and all parameters $a_{m,i}$. The residual error of each pixel k on each object m are computed as:

$$r_{m,k} = f_m(x_k) - y_k$$

The "E-step" in the EM algorithm assumes that the parameters are given when we calculate the probability. Initially all parameters are randomly assigned. The probability of a point k belonging to an object m over Gaussian probability distribution is given as:

$$P(k|m) = p_{m,k}/(p_{1,k} + p_{2,k} + \ldots + p_{M,k}),\ p_{m,k} = \exp(-r_{m,k}{}^2/\sigma_n)$$

The M-step of EM algorithm takes the probability, and re-estimates the parameters using weighted least-squares. That is, the following weighted error function on the model parameters is minimized:

$$E_m = \Sigma\, P(k|m) r_{m,k}{}^2$$

The E-step and M-step are repeated until they converge to a solution.

Object motion is an important feature. The theorem described in Section 2.3.2 is usable. But it needs a lot of computation. A simpler way is only to compute edges motion. While the pixel number of edges is much fewer than a picture, it gets good speed improvement. By edge motion, we can also know about how an object is rotated and its velocity.

Object distance is detected by stereo estimation. It needs two cameras work as human eyes. By lens focus length, camera distance and object location difference in two images, the distance can be calculated.

All the above image processing algorithms need similar calculations as video processing, such as filtering and transform operations composed of multiplications and additions.

2.5 Cryptography

Digital Video Broadcasting (DVB) system supports Video-on-Demand. Content provider only allows an authorized user to watch this video. The DVB common interface (DVB-CI) standardizes a conditional access module (CAM) for DVB receiver to adapt content with kind of cryptography. Equipped with a PCMCIA card attached on DVB-CI, the receiver can send the video stream into the card, and get a decrypted stream which will be sent to the MPEG demultiplexer.

Many *cryptographic* algorithms had been introduced. They can be divided into two categories: *public-key* and *symmetric-key*.

In *public-key cryptosystem*, the keys to encrypt and decrypt a context are different. To use it, a member needs a smartcard as the electronic identifier. The smart card generates a pair of keys from a random number. A private key is stored in the card, and a public key released to public domain. Anyone who wishes to send a message to this member can encrypt the message by the public key. Only the smartcard with the provided private key can decrypt this message. Public-key cryptographic is based on a complex arithmetic which needs very intensive computation, thus public-key is only used on authorization. *RSA* is one example of the most popular public-key algorithms.

Symmetric-key cryptosystem uses the same key to encrypt and decrypt message. Thus it needs a public-key cryptosystem to exchange the key. *Data Encryption Standard* (DES) and *Advanced Encryption Standard* (AES) are two examples of such standard in use. We will discuss briefly these three standards in the following subsections.

2.5.1 RSA

RSA algorithm was described in 1977 by Ron Rivest, Adi Shamir, and Leonard Adleman at MIT. The derivation of RSA is based on the following number theorems.

1. *Prime number Theorem*
   Gauss conjectured the prime number theorem in 1793. He defined a new function $\pi(x)$ to denote that the number of primes does not exceed x, where x is a positive number. The theorem is described as follows:
   The ratio of $\pi(x)$ to $x/\ln(x)$ approaches one as x grows without bound. That is:

$$\lim_{x\to\infty} \frac{\pi(x)}{\frac{x}{\ln x}} = 1$$

   The prime number theorem reveals two things for us:

   (1) If we randomly write two numbers, the one with more digital numbers is with the less probability to be a prime.
   (2) The number of big prime numbers is more than the number of small prime numbers.

2. *Euler Theorem*
   If m is a positive integer and a is an integer with $(a, m) = 1$, then

$$a^{\psi(m)} \equiv 1(\bmod m)$$

Here $\psi(m)$ is an Euler function which is defined to be the number of positive integers not exceeding m and being relatively prime to m.

3. *Fermat's Little Theorem*
If p is prime and a is a positive integer, p is not a factor of a, then

$$a^{p-1} \equiv 1(\bmod p)$$

Fermat's Little theorem is a special case of Euler theorem, we can get the modular exponentiation answer quickly when we apply this theorem.

The *RSA cipher algorithm* can be divided into the following eight steps.

(1) Select two large prime numbers p and q and compute $N = p \times q$.
(2) Compute $\psi(N) = (p-1)(q-1)$.
(3) Select an odd integer e such that e and $\psi(N)$ are coprime.
(4) Compute d, $d \times e = k \times \psi(N) + 1 \equiv 1(\bmod \psi(N)) => d = e^{-1} \times (\bmod \psi(N))$.
(5) Publish the pair $P = (e, N)$ as an RSA public key.
(6) Keep secret the pair $S = (d, N)$ as an RSA secret key.
(7) Encrypt ciphertext $C = M^e(\bmod N)$, where $M =$ plaintext. And
(8) Decrypt $C^d(\bmod N) = M^{e\times d}(\bmod N) = M^{k\times\psi(N)+1}(\bmod N) = M(\bmod N)$.

The RSA cipher system security is based on the difficulty to find the factor for a large N. Typically N is 1024-bit long today. The choice of p and q should be strong prime to ensure the security.

To computation of modulo exponential $C = M^e(\bmod N)$ is to recursively apply modulo multiplication, as follows:

```
C=M;
for(i=log$_2$e-1; i>=0; i--) {
     C=C×C(mode N);
     if (e$_i$) C=C×M(mod N); //e$_i$=bit i of e
}
```

The modulo multiplication requires large number division which is a high cost computation. Peter Montgomery introduced an efficient algorithm to reduce the cost [21]. The algorithm is described as follows to compute the equation $Z = A \times B \times 2^{-k}$ (mod N):

```
Z=0;
for(i=0; i<k; i++) {
    if (Z+a$_i$×B is even) Z=(Z+a$_i$×B)/2; //a$_i$=bit i of A
    else  Z=(Z+a$_i$×B+N)/2;
}
```

Here k is bits of A. While N is odd number, the divided-by-2 in both `if` and `else` paths will not lose bit. Above algorithm computes:

$$Z = (a_0 2^{-k} + \ldots + a_{k-1} 2^{-1}) \times B + t \times N = A \times B \times 2^{-k} + t \times N \equiv A \times B \times 2^{-k} (\mathrm{mod}\, N)$$

Here t is a number derived by even test. Apply *Montgomery's algorithm* into modulo exponential computation, the 2^k factor should be adjusted, as shown in the following:

$$(A \times 2^k)^*(B \times 2^k) \times 2^{-k} (\mathrm{mod}\, N) = A \times B \times 2^k (\mathrm{mod}\, N)$$

The modulo exponential algorithm can thus be changed as follows:

```
$C$=$M_k$=$M$×$2^{2k}$ ×$2^{-k}$ (mod $N$);
for($i$=log$_2$$e$-1; $i$>=0; $i$--) {
    $C$=$C$×$C$×$2^{-k}$(mode $N$);
    if ($e_i$) $C$=$C$×$M_k$×$2^{-k}$(mod $N$); //$e_i$=bit $i$ of $e$
}
$C$=$C$×$2^{-k}$(mod $N$)
```

2.5.2 DES

Data Encryption Standard (DES) is defined by Federal Information Processing Standard (FIPS) for United States in 1976 [22]. Its algorithm is shown in Fig. 2.27, where the plaintext is decomposed into many 64-bit blocks and the key size is 56-bit. The algorithm is composed of many permutation, substitution and exclusive-or operations. If bit-level permutation uses wire connection, the hardware implementation cost is low. But since it needs many shift operations in software implementation, without the support of hardware permutation instructions, its running speed is very slow.

2.5.3 AES

Advanced Encryption Standard (AES) becomes new FIPS standard in 2001 [23]. AES operations are not regular integer, they are computed on 8-bit Galois Field GF(2^8). The basis polynomial is f(x) $= x^8 + x^4 + x^3 + x + 1$, or hex number f(2) $= 0x11$B while $x = 2$. If a 8-bit value v is multiplied with 3 in GF(2^8), its behavior is shown as the following code, where $\odot$ and $\oplus$ respectively represent multiplication and addition in GF(2^8).

```
$w$ = $v$⊙$2^1$ = ($v$&0x80)?(($v$<<1) mod f(2)) : ($v$<<1)
              = ($v$&0x80)?(($v$<<1) xor 0x11B) : ($v$<<1)
$v$⊙3 = $v$⊙($2^1$⊕$2^0$) = ($v$⊙$2^1$)⊕($v$⊙$2^0$) = w xor v
```

The hardware implementation of GF(2^8) operation is only bit test and exclusive-or without carry propagation, so its delay is low. Its software implementation is more complex than regular integer, most implementations use lookup table instead

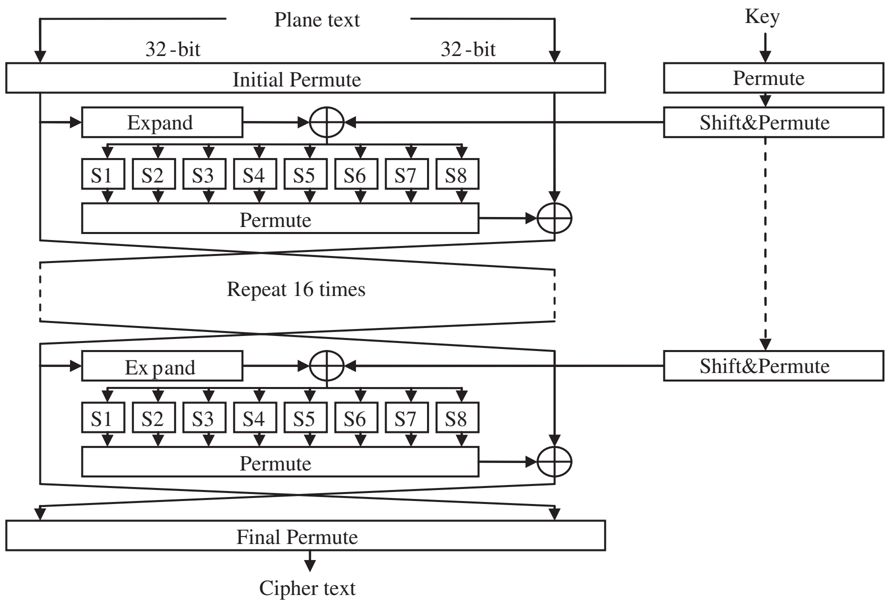


Fig. 2.27 DES encryption flow

of direct computation. The use of GF(2^8) provides a simple but good non-linearity representation that increases the difficulty to attack by algebraic analysis.

Typical AES block size is 128 bits, formed as a 4×4 byte array. Its key size can be chosen from 128, 192 or 256 bits. The AES algorithm is composed of 5 functions, `SubBytes, ShiftRows, MixColumns, AddRoundKey, and KeyExpansion`. The `KeyExpansion` function expands original key for each round by a recursive substitution. `AddRoundKey` function combines the 4×4 array with substituted round key. The `SubBytes` function substitutes each byte in the 4×4 array. The substitution is defined as an S-box which is derived from the multiplicative inverse over GF(2^8). The `ShiftRows` function cyclically shifts the bytes in each row by a certain offset. The `MixColumns` function combines the four bytes of each column using an invertible linear transformation. Each column is treated as a polynomial over GF(2^8) and is then multiplied modulo $x^4 + 1$ with a fixed polynomial $c(x) = 3x^3 + x^2 + x + 2$ (Fig. 2.28).

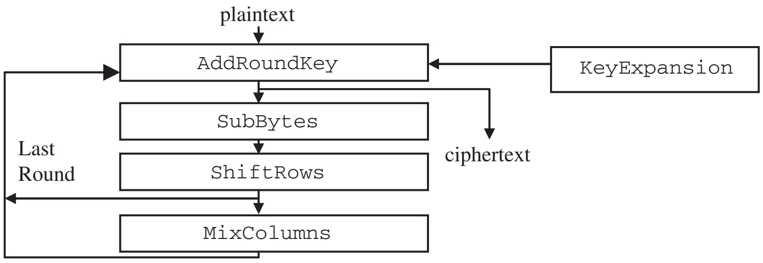


Fig. 2.28 AES encryption flow

2.6 Digital Communication

Software Defined Radio (SDR) offers a programmable and dynamically reconfigurable method of reusing hardware to implement the physical layer processing for multiple communications systems.

A wireless communication system is partitioned into three parts according to its signal frequency. The *Radio Frequency* (RF) part processes the carrier wave frequency. Various licensed/un-licensed bands have been used for different applications, such as 1.8 GHz for GSM, 2.4 GHz for 802.11b, and so on. The *Intermediate Frequency* (IF) part processes on lower frequency. The *Baseband* part process on digital data, it connects to IF through ADC/DAC.

The major components of a baseband include modulation/demodulation, equalization, and clock/data recovery. By the increasing DSP ability, many baseband functions can now be implemented in DSP software. In low-IF system such as Bluetooth, some IF functions can even be moved to DSP.

By IF selection, a wireless RF front-end receiver can be built using one of the following three architectures: super-heterodyne, low-IF or zero-IF. Figure 2.29 shows these architectures.

The *Low Noise Amplifier* (LNA) as shown in Fig. 2.29 boosts the weak channel above the noise floor of the mixer. The *mixer* shifts center frequency by multiplying a sinusoid wave which is generated by a *Local Oscillator* (LO). In super-heterodyne, $\omega_{LO} = \omega_{RF} - \omega_{IF}$. In low-IF, $\omega_{LO} = \omega_{RF}$ – (a small value). In zero-IF, $\omega_{LO} = \omega_{RF}$. A mixer can up-convert (or down-convert) a signal by performing the following equation:

$$\cos \omega_{RF} \times \cos \omega_{LO} = (\cos(\omega_{RF} + \omega_{LO}) + \cos(\omega_{RF} - \omega_{LO}))/2.$$

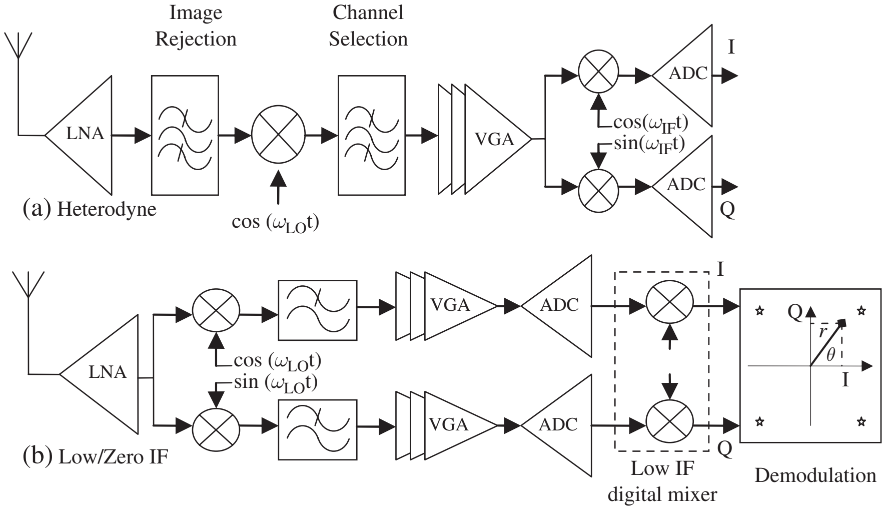


Fig. 2.29 Receiver front-end architectures

The later item represents the signal we wish, and prior item is a high frequency signal that should be removed by low-pass filter. The image signal whose frequency is $\omega_{RF} - 2\omega_{IF}$ will be shifted to $-\omega_{IF}$. It is the same frequency with 180° phase shift, that will affect the IF signal, so it should be filtered out before entering mixer.

Why do we need an intermediate frequency IF? The main reason is the DC offset and flicker noise of the local oscillator LO. A signal is composed of two components I and Q which are generated by respectively mixing the LO wave with two orthogonal cosine and sine waves. The LO DC offset and flicker noise will make the two mixers in Fig. 2.29(b) mismatch and get imbalanced I and Q values. The super-heterodyne architecture avoids LO non-ideality effect, but needs better analog circuit design knowledge. Low-IF is a trade-off solution that selects a low intermediate frequency which can do IF mixing by digital circuit.

Each standard defines abundant of modulation methods. *Modulation* can be performed on frequency, amplitude (r), phase (θ), or their combination. In Fig. 2.29(b), the demodulation stage shows a QPSK example. Four legal positions are defined at $r = 1$ and $\theta = \pm 45°$ and $\pm 135°$. The demodulation stage should decide the received I/Q values belong to which position, and adjust receiver parameters to reduce the phase shift.

The purpose of modulation is to reduce signal frequency but have the bit rate remain the same. Radio wave propagates on air. It will be absorbed by particles in air that causes fading effect and non-linear channel frequency response. It may be reflected by building which causes multi-path effect. Other waves can be received by same antenna. When transmitter or receiver is moving, Doppler Effect is involved. Also the spreading power of previous symbol will affect it as the *Inter-Symbol-Interference* (ISI) problem. Figure 2.30(a) shows the waveform of a non-modulated signal; it has large distortion by above effects. Figure 2.30(b) is modulated by ASK, each symbol carries 2 bits. The signal frequency is reduced to half, so it is better on channel frequency response. But it has 4 legal voltage positions, the noise margin is reduced.

Orthogonal Frequency Division Modulation (OFDM) is a way to increase bandwidth. It is widely used in 3G and newer communication standards. OFDM uses many overlapped orthogonal sub-carriers to carry more bits in one symbol. The waveform is combined as shown in the following equation. IFFT (Inverse Fast Fourier Transform) is used to generate waveform, and FFT used to demodulate a received signal.

$$s(t) = \Sigma d_k \exp(-j\omega_k t),\ k = 0 \text{ to } N-1,\ \omega_k = 2\pi k/NT$$

In addition to ISI, OFDM has to overcome the *inter-channel interference* (ICI) problem. Figure 2.31 shows the spectrum of a single signal channel and OFDM

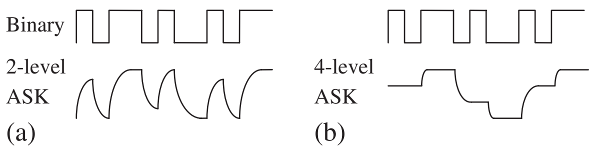


Fig. 2.30 Waveform of ASK modulation

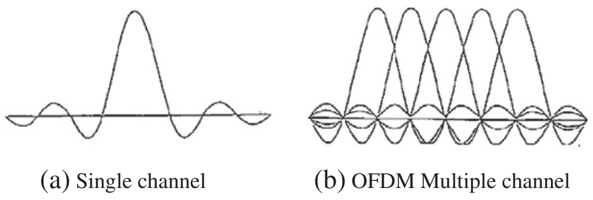


Fig. 2.31 OFDM spectrum

multi-channel. To combat ICI, the cyclic prefix is attached on original symbol to use as the guard interval to maintain orthogonal.

To solve transmission channel non-idealities, baseband needs to perform some digital Finite Impulse Response (FIR) or Infinite Impulse Response (IIR) filtering.

Equalizer inverses the fading effect and channel frequency response. Most communication protocol provides a preamble sequence for receiver to detect channel characteristic to decide the equalizer parameter before data transmission.

Match filter finds the best sampling point with best signal-to-noise ratio (SNR). Facing the multi-path effect, a wave will be received multiple times with different delays. Match filter samples a symbol at multiple phases, and outputs the one with the best SNR.

Clock Data Recovery (CDR) adjusts Local Oscillator frequency drift by detecting symbol transition edge. The frequency drift will cause the I/Q rotation in a fixed rate. CDR can apply a rotate computation on I/Q to opposite direction to compensate the frequency.

To reduce ISI effect, transmitter reshapes a square impulse to a smooth one by *Nyquest Filter*. Two filter coefficients are popular. One is Gaussian distribution coefficient, another is raised cosine filter.

When a channel is not guaranteed to be good enough, some received bits may be in error. Communication protocol requires a method to detect this error and correct it. When a channel is really bad, re-transmission will never succeed, or it will violate real-time constraint, then an *error correction coding* with redundant information is needed. There are two information coding theories, block coding and convolution coding.

Block coding processes information in block. *Hamming code* is the most popular in use. For an n-bit data, $\log_2 n + 1$ bits are appended to the data for error detection and correction.

Convolution coding distributes information into a continuous sequence. Figure 2.32 shows a generic form. The input sequence is latched into 6 stages, and

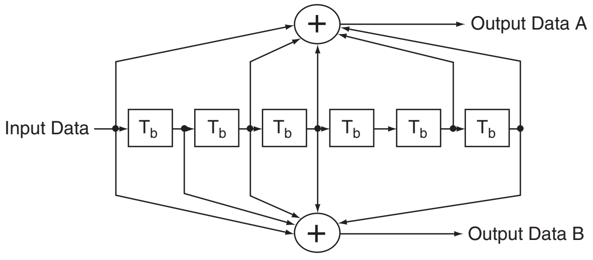


Fig. 2.32 Convolution coding

summarized by two equations to get 1/2 code rate. Other structures can get 2/3 or 3/4 code rates. Convolution coding has better coding gain than block coding in the same code rate. The most widely used convolution coding is *Viterbi* algorithm which is an elegant method for performing maximum likelihood.

2.7 Multimedia Instruction Set Design

Above sections had discussed most interesting multimedia applications. We summarize with some special features from these applications to show the need of building specific instructions in an SIMD processor.

The first feature is low data precision and large data parallelism. Image pixels are typically stored with an 8-bit precision and audio samples stored with a 16-bit precision. The ADC used in Fig. 2.29 needs mostly a 10-bit precision. AES is computed on an 8-bit Galois Field GF(2^8). Only RSA requires larger bits of precision.

All multimedia applications contain large data parallelism. In video coding, SAD operations are applied to every macroblocks with different search distances (m, n). Transform, Quantization and De-blocking filter basically use matrix operations. In linear algebra, row or column elements of a matrix can be processed concurrently. Image processing mostly consists of matrix operations, which can thus be parallelized. In AES, `SubByte` operation can be parallelized on all bytes, `ShiftRow` can be parallelized in row-major order, and `MixColumn` parallelized in column-major order. In digital communication, I and Q components composed as a vector could be computed simultaneously. In OFDM, data can be extracted from all sub-channels in parallel.

The low data precision and large data parallelism features induce the idea of *subword-parallel*, *single instruction multiple data* (SWP-SIMD) design. A traditional SIMD processor uses many ALUs to compute multiple data in a single instruction. While multimedia data resolution is low, these multiple data can be packed into a single register and computed in a single 64-bit or 128-bit ALU.

Fixed-point multiplication operation is common in multimedia algorithms, such as DCT in video coding and filters in digital communication. Fixed-point multiplication needs a right shift after multiplication to keep the most-significant bit.

Permutation is frequently used in multimedia algorithms. In DCT or FFT, data are often re-arranged in a so-called *butterfly* order. DES is based on complex permutation. In an SWP-SIMD processor, since these data are in a register, re-arranging them by a left/right shifter is very difficult. In general, a 64-bit register can have $64! = 10^{89}$ kinds of permutations. Fortunately, most applications use only a limited subset of permutations. Thus, in PLX, we have only to implement PERM, a configurable permutation instruction unit, with an 8-bit *palette* argument that allows us to select a required PERM instruction from the 256 pre-defined permutations.

Performances in video coding and image processing are limited by their memory latency. For example, an H.264 code using five reference frames needs to buffer five pictures in memory. When using image recognition to track a feature in a series of pictures, the picture size is often too large to put in an on-chip memory; thus, an external memory is required. Today external memory speed is much slower than processor speed. *Memory latency* dominates the performance in these applications. Techniques that can reduce memory latency are often used in a multimedia processor. These techniques include caching, perspective preload, and multi-threading.

Floating point unit is another difficult choice. A floating point processing unit takes a large chip area and consumes more power than an integer processing unit. For example, 3-D graphics needs a floating point processing unit to compute object rotation. In video coding, image processing and digital communication, since data precision is limited by human sensors' sensitivity and channel quality, a small amount of precision loss is acceptable. That is why we can perform the counter-part multimedia operations using an integer processing unit to save cost and power.

PLX [1] has been developed by Professor Ruby Lee at Princeton University. The main feature of PLX is that it is a native *subword-parallel single instruction multiple data* (SWP-SIMD) processor [2]. Its vector function unit supports 8/16/32/64 subword widths, and its scalar function unit is the 64-bit subword subset in a vector unit. A typical multimedia code contains many scalar operations, such as loop counter or memory index. A native SWP-SIMD can execute scalar and vector operations in the same core to reduce scalar-vector communication overhead. Thus, it is very suitable to be used in the above discussed video and image processing, cryptography, and communication applications.

In this book, we will depict the hardware/software codesign methodology and tools applied in the development of PLX, the designs of a parallel compiler, a profiling tool, and an OS kernel for PLX. Details of these tools or processor designs will be discussed in the following chapters.

Chapter 3
System Level Design

The goal of a system-level design methodology is to decrease design cost and design time. Firstly, the complexity of a modern system does not allow us to describe its implementations directly. Furthermore, it is difficult to create derivative implementations with different functions or different architectures, because functions and architectures cannot be extracted easily from implementations for reuse. Therefore a separation of function, architecture and implementation is necessary when designing a system. Design activities are needed to combine different functions and architectures for an implementation to decrease design time and design cost.

To be able to separate function, architecture and implementation in a system design flow, different abstraction levels need to be defined. The idea is to gradually confront designers with implementation details such as timing and data representations. Abstraction levels are used to describe the functionality of the target design by mathematical equations and/or algorithms. It then goes through behavior synthesis processes to generate register transfer level (RTL) circuit (Fig. 3.1).

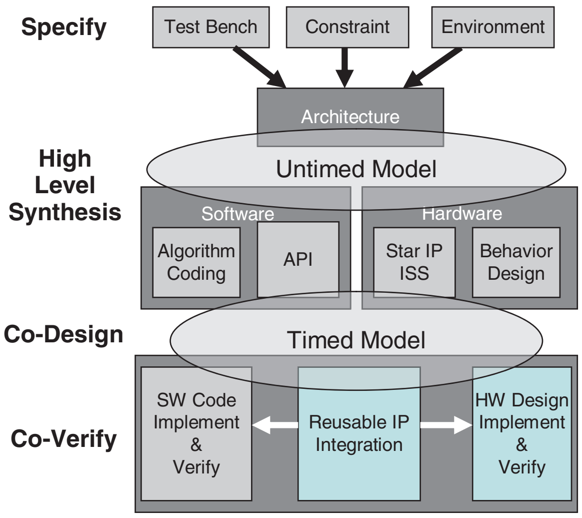


Fig. 3.1 System level design

3.1 Abstraction Levels

A design can be described on different levels of abstraction. Raise the abstraction level is always a trade-off between the speed and accuracy of a potential simulation model. Function level only captures the algorithm regardless of the implementation details, an algorithmic model has a huge advantage in its high simulation speed. On the other end, RTL simulation accuracy is fidelity to real implementation. But it is too expensive to pay due to its lengthy simulation time.

3.1.1 Algorithm Level

The motivation for introducing this *algorithmic level* of abstraction is to quickly obtain a function to determine what the system is supposed to do, without making architecture assumptions. Hence there is the potential to reuse functions either to create derivative functions, or to synthesize different implementations with different architectures.

Algorithm Level contains two main subjects: *algorithm specification* and *data communication*.

In the algorithm specification, an executable functional specification of the algorithm is created. It may be a C/C++ or Matlab code. This executable specification is used to check the validity of the algorithm. The simulation in this design step is sequential, which has no timing information and uses a single thread of control. The simulation speed is high due to lack of timing and architecture details.

Profiling techniques are used to obtain an initial estimate of the computational load of the different functions and the amount of data transfer between them.

Code inspection is used to estimate the amount of flexibility required for each of the functions. The results of both, code inspection and profiling, are used as input for a task and, in a later stage, for hardware/software partitioning.

By the executable algorithm specification, a golden reference model is generated for verification throughout the whole flow.

With the design constraints and requirements and a suitable architecture template in mind and the results from the previous algorithm design step, the system is partitioned into tasks that perform functions and channels through which data are communicated between these tasks. With a multi-threaded simulation tool, the communication load on the channels and the computation load on the tasks can be analyzed. If necessary, the system can be repartitioned to meet the constraints and requirements.

3.1.2 Architecture Level

The motivation of this *architecture level* is to quickly find an efficient architecture implementation. Efficiency can be defined in terms of power, timing, area, *etc*. To be able to quickly evaluate the efficiency of alternative implementations, we want to avoid the effort of making them in detail. For example the decision to base an implementation on a message-passing or a shared-memory architecture leads to two alternative implementations.

Transaction Level Modeling (TLM) is developed for architecture level design and exploration. Literally a transaction is the exchange of goods, services or funds; or a communicative action or activity involving two parties or things that reciprocally affect or influence each other. Both meanings have two ingredients, exchange/communication and goods/influence. In an electronic system the goods or influence can be considered as the computation or the effect of the computation. There are many discussions regarding TLM over the years, here we use only definitions, terminologies and libraries developed by the OSCI TLM Working Group (TLM WG) [24].

Although TLM includes computation and communication, OSCI TLM 1.0 and 2.0 discuss only the communication part. In the physical form, a transaction is a payload, the data structure that passed between modules. By definitions from TLM WG, we have following four abstraction levels (or called modeling styles):

(1) *Un-Timed* (UT): A modeling style in which there is no explicit mention of time or cycles, but includes concurrency and sequence of operations. In the absence of any explicit notion of time as such, the sequencing of operations across multiple concurrent threads must be accomplished using synchronization primitives such as events, mutexes and blocking FIFOs. Some users adopt the practice of inserting random delays into untimed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style.
(2) *Loosely Timed* (LT): A modeling style that represents minimal timing information sufficient only to support features necessary to boot an operating system and to manage multiple threads in the absence of explicit synchronization between those threads. A loosely timed model may include timer models and a notional arbitration interval or execution slot length. Some users adopt the practice of inserting random delays into loosely timed descriptions in order to test the robustness of their protocols, but this practice does not change the basic characteristics of the modeling style.
(3) *Approximately Timed* (AT): A modeling style for which there exists a one-to-one mapping between the externally observable states of the model and the states of some corresponding detailed reference model such that the mapping preserves the sequence of state transitions but not their precise timing. The degree of timing accuracy is undefined.
(4) *Cycle Accurate* (CA): A modeling style in which it is possible to predict the state of the model in any given cycle at the external boundary of the model and thus to establish a one-to-one correspondence between the states of the model and the externally observable states of a corresponding RTL model in each cycle, but which is not required to explicitly re-evaluate the state of the entire model in every cycle or to explicitly represent the state of every boundary pin or internal register. This term is only applicable to models that have a notion of cycles.

3.1.3 Behavior Level

All abstraction level above RTL can be called behavior level. Here we specify that architecture is implemented and interface is pin-accurate. This abstraction level captures the details of the interfaces and the input/output (I/O) functionality including a full or partial specification/modeling of their timing. The communication among blocks is carried out by signals.

The behavior level introduces some freedom in how operations and I/O are scheduled by only partially constraining the cycle-by-cycle behavior of I/O. Registers are not explicitly defined, but instead are determined by synthesis. Storage requirements are dependent on how operations are scheduled: registers are used to store values that are used one or more cycles after the cycle in which they are generated. Storage of arrays may be mapped to memories or to registers. The specification of behavior is in the form of an implicit-state machine rather than the explicit-state machine generally used for RTL. In an implicit-state machine, there is no explicit state variable that is used to select what behavior is executed next. Instead, the behavior consists of a process that is sensitive to signal events. The process uses language constructs such as `loop`, `if-then-else` and `case` to specify conditional behavior and wait statements to specify cycle timing among sets of output assignments.

The behavior level description is able to be synthesized into RTL design by the technique of behavioral synthesis or high-level synthesis. The verification methodology of the generated specifications against the behavioral specification is more complex than the RTL level since the cycle-by-cycle behavior may be changed by high-level synthesis.

The untimed and timed TLM are models tailored for distinct purposes. The ultimate goal is to create a unique platform that simulates two different models according to user needs. The *untimed TLM* is an architectural model targeted specifically at early functional software development and functional verification where timing annotations are not compulsory conditions. The high simulation speed is the objective of this model. Since the untimed TLM serves primarily programmers, it is hence given another name as *programmer view* (PV).

On the other hand, the *timed TLM* is a micro-architectural model containing essential time annotations for behavioral and communication specifications. It is relatively a less abstract model located lower in the SoC design flow. The focus of timed TLM is the simulation accuracy required by real-time embedded software development and architecture analysis. Hence, the timed TLM is also known as *programmer view plus timing* (PVT).

Thus, the behavior level has following abstract levels:

(1) Communicating Processes (CP)

- Behavior is partitioned into parallel processes.
- Communication is implemented by point-to-point.

(2) Communicating Processes + Timing (CPT)

- CP model annotated by timing information.

(3) Programmer View (PV)

- Behavior model is register accurate at the low-level
- Programmer API
- Point-to-point communication are refined to bus or NoC models
- Sequenced, but untimed model of computation

(4) Programmer View + Timing (PVT)

- PV + timing annotations (cycle estimates)
- Refined communication model
- Communicating processes are mapped onto HW blocks

(5) Cycle Callable (CC)

- System behavior model with cycle true timing, no pin-level details

(6) Register Transfer (RT)

- Synthesizable model, with pin-level details

RTL is widely used and was traditionally written in HDL languages. At this abstraction level, finite states and registers used for state transfer are determined. The verification methodology of the output from RTL synthesis against the reference RTL specification is well defined for both the combinational and sequential hardware. For instance, IEEE Std 1076-2004 defines this for VHDL.

3.2 Algorithm Level Verification

There are more than 70% of errors in a design found to be functional. A design has to be functionally correct and fully implemented as stated in the specification. As modern design is getting more and more complex, the testbench for the design and properties/assertions of the design grow bigger and more complex at the same time. This implies those testbenches, properties and assertions manually written in advance are error-prone, as long as there is something done manually. The ambiguities introduced by abstraction which is said to be an important concept to deal with the increased complexity depreciate the values of a high-level design language.

At the moment all existing methods are still using a certain degree of human intervention to deal with practical tricky problems, we realize that a couple of things are required to ensure fewer bugs in a design. First, fewer things, no matter test pattern generation or properties/assertions writing done manually in each step of the design flow, are critical. Second, a high-level definition/language with fewer ambiguities is needed. Last, in the case we need to verify a losable implementation which most multimedia application is, we expect that a verifier can tell how much precision was lost in a losable implementation.

In this section, we propose a methodology where neither testbenches, nor assertions/properties writing are needed to verify and tell the degree of error that an implementation will cause. The specification defined in mathematic expressions is used as golden model and thus eliminates the need of human interventions in the design flow. An implementation is translated into a form of mathematic expression and then the equivalence between the implementation and the golden model is checked by Mathematica. If there is something different between the two expressions, the error can be easily calculated since both of them are just mathematic expressions.

This novel methodology incorporates the technologies of reverse engineering, theorem proving and the advantage of unambiguous mathematical expressions. As one problem can be solved by a code or a digital hardware implementation, it can be modeled mathematically and verified by our proposed methodology. Of course, the error caused by an implementation of a certain specification/algorithm can then be predicted by the mathematical expressions derived by taking the advantage of the precision of mathematical expressions. To precisely estimate the error of an implementation is what we are going to do next.

The SAD algorithm implemented on our designed SIMD PLX platform is shown to be verified by the proposed methodology. An SIMD platform provides higher computing power by executing multiple data simultaneously, but this feature is making the design and verification harder. We aim at helping designers with an efficient methodology for verifying and evaluating the performance of our SIMD PLX platform design. Using Mathematica, a computer algebra system (CAS), and reverse engineering techniques, the correctness and errors accumulated in running multimedia applications on this PLX platform can be precisely evaluated, which would not be easily done without human interventions before. The proposed methodology can be easily automated and adapted to other platforms.

While a system is being developed, the designer always needs to know as soon as possible whether the designing embedded software or hardware faithfully achieves the design goal. However, the designer cannot be free from time-consuming properties/assertions writing nor from stimuli or test pattern generation, no matter a formal verifier [25] (which uses a CAS as formal verifier) or a simulation methodology is going to be applied. Especially when we would like to know how the error of a losable multimedia implementation is, we have to deal with running high order logic (HOL) with a theorem prover [26, 27] or calculating it manually. Thus, a simple and efficient methodology able to verify the function and evaluate the errors of an implementation is desirable.

In the following subsections, we will first introduce the techniques and then use an algorithm to show the detail of our verification and error prediction methodology.

3.2.1 Algebraic Simulation

Some operations in an implementation which move or arrange data only without mathematic operations can be easily verified by algebraic simulation. The process is illustrated by an assembly code which performs a matrix transpose on PLX. Every data rearrangement and memory operation is modeled directly into a Mathematica instruction. For example, the instruction `load.sw.Rd, Rs1, imm13` loads `sw` sub-words in memory addresses starting at “offset” with `Rs1` = offset, `sw` = 4 (or 8) and is modeled as follows in Mathematica:

```
R[1,1],R[1,2],R[1,3],R[1,4]=
        Table[a[1,j],j,1,4]
```

Moreover, the transpose of a 4×4 matrix in PLX assembly code as shown in Fig. 3.2 can be modeled in Mathematica as shown in Fig. 3.3.

```
mix.2.l Rt1, R1, R2   mix.1.r R3, Rt1, R1
mix.2.r Rt2, R1, R2   mix.1.l R1, Rt1, R1
mix.2.l R1, R3, R4    mix.1.r R4, Rt2, R2
mix.2.r R2, R3, R4    mix.1.l R2, Rt2, R2
```

Fig. 3.2 A 4×4 Matrix transpose in PLX assembly

```
{R[1,1],R[1,2],R[1,3],R[1,4]} =
 {R[3,1],R[3,2],R[3,3],R[3,4],R[4,1],R[4,2],R[4
 ,3],R[4,4]} [[{1,5,3,7}]]
{R[2,1],R[2,2],R[2,3],R[2,4]} =
 {R[3,1],R[3,2],R[3,3],R[3,4],R[4,1],R[4,2],R[4
 ,3],R[4,4]} [[{2,6,4,8}]]
{R[3,1],R[3,2],R[3,3],R[3,4]} =
 {R[5,1],R[5,2],R[5,3],R[5,4],R[1,1],R[1,2],R[1
 ,3],R[1,4]} [[{3,4,7,8}]]
{R[1,1],R[1,2],R[1,3],R[1,4]} =
 {R[5,1],R[5,2],R[5,3],R[5,4],R[1,1],R[1,2],R[1
 ,3],R[1,4]} [[{1,2,5,6}]]
{R[4,1],R[4,2],R[4,3],R[4,4]} =
 {R[6,1],R[6,2],R[6,3],R[6,4],R[2,1],R[2,2],R[2
 ,3],R[2,4]} [[{3,4,7,8}]]
{R[2,1],R[2,2],R[2,3],R[2,4]} =
 {R[6,1],R[6,2],R[6,3],R[6,4],R[2,1],R[2,2],R[2
 ,3],R[2,4]} [[{1,2,5,6}]]
```

Fig. 3.3 Matrix transpose modeled in Mathematica

The result of this Mathematica code is shown in Fig. 3.4, where the matrix has obviously been transposed. This algebraic simulation example shows how variables are represented symbolically and reveals how operations are applied on variables and instructions executed in Mathematica.

```
R1 = {a[1,1],a[2,1],a[3,1],a[4,1]}
R2 = {a[1,2],a[2,2],a[3,2],a[4,2]}
R3 = {a[1,3],a[2,3],a[3,3],a[4,3]}
R4 = {a[1,4],a[2,4],a[3,4],a[4,4]}
```

Fig. 3.4 Result of matrix transpose

3.2.2 Algebraic Analysis

For those implementations that perform complex `for-loops` and/or `while-loops`, all conventional simulation methodologies are not suitable. Instead, an algebraic analysis is called for. In the following, algebraic analysis of an assembly implementation of the Sum-of-Absolute-Differences (SAD) algorithm on a PLX platform is illustrated.

The SAD is a criterion used in block-based matching motion estimation algorithms to gauge the similarity between a given macroblock in the current frame and a corresponding macroblock in a reconstructed reference frame. The displacement between these two macroblocks is used to find a motion vector candidate. For a K×L macroblock, one has:

$$SAD\,(m,n) = \sum_{i=0}^{K-1}\sum_{j=0}^{L-1} |C(i,j) - R(m+i, n+j)|,$$

where $C(i,j)$ is the luminance value of a current frame pixel and $R(i,j)$ is the luminance value of a reference frame pixel. Argument (m,n) is the displacement between these two blocks.

First, the SAD assembly code listed in Fig. 3.5 is parsed and translated by running the symbolic verification algorithm as shown in Fig. 3.6. Mathematical expressions are derived after the loops and subroutine calls.

The initial values and upper bounds derived by loop unwinding are replaced by the real values specified in the code as:

```
Table [For[j=1; b[m,n]=0, j<4,
 For [i=1, i<4,
      b[m,n]=b[m,n]+Abs[C[i,j]-R[m+i,n+j]];
      i++]; j++];  b[m,n], {m,1,16}, {n,1,16}]
```

This derived expression is compared to the very original design goal:

```
Table[SAD[m,n] = ∑_{i=1}^{4} ∑_{j=1}^{4} Abs[C[i,j]-R[m+i,n+j]],
                 b[m,n], {m,1,16}, {n,1,16}]
```

If the obtained comparison result is true, the design correctness is verified. Since loops and subroutines are analytic unwound [28], the algorithm complexity is O(number of instructions).

3.2.3 Error Evaluation

Once an implementation is found to be "losable," the range of errors will become a matter of concern. The maximum possible absolute/relative error is easy to figure

Asembly codes	Extracted expression
ME_8bitsat proc	
;//unsigned char *R21=cur;	
;//unsigned char *R22=ref;	
;//unsigned char *R23=sad;	
loadi.z.0 R12,0	
loadi.z.0 R13,0	S=0
loc_me_8s_1:	
loadi.z.0 R7,0	
loadi.z.0 R8,0	
loadi.z.0 R14,0	t=0
loc_me_8s_2:	
load.8.update R3,R21,0	R3[j]=cur[i,j]| $_{j=0\sim7}$
addi R21,R21,8	j=j + 8
load.8.update R4,R21,0	R4[0~7]=cur[i,8~15]
addi R21,R21,width-8	j=j + width-8 ,,=> i=i+1
:	:
psub.1.u R18,R3,R5	R18[j]=cur[i,j]-ref[i,j]| $_{j=0\sim7}$
psub.1.u R19,R5,R3	R19[j]=ref[I,j]-cur[i,j]| $_{j=0\sim7}$
padd.1.u R18,R18,R19	R18[j]=|cur[i,j]-ref[i,j]| $_{j=0\sim7}$
padd.1.u R7,R7,R18	R7[j] = SUM|cur[i,j]-ref[i,j]| $_{j=0\sim7,t=0\sim3,,i=t+4s}$
addi R14,R14,1	t++
cmpi.eq R14,4,P1,P2	end of for(t=0;t<4;t++)
P2 jmp loc_me_8s_2	
Add_4byte_u R11,R7	R11[0]=SUM(SUM|cur[i,j]-ref[i,j]|)| $_{i=(0\sim3)+4s,\ j=0\sim3}$
deposit R12,R11,0,8	R12[0]=R11[0]
extract R11,R11,40,8	R11[0]=SUM(SUM|cur[i,j]-ref[i,j]|)| $_{i=(0\sim3)+4s,\ j=4\sim7}$
deposit R12,R11,8,8	R12[1]=R11[0]
:	:
store.4 R12,R23,0	[R23+4s+k]<-SUM(SUM|cur[i,j]-ref[i,j]|)| $_{i=(0\sim3)+4s,\ j=(0\sim3)+4k,\ k=0\sim3,,\ s=0\sim3}$
addi R23,R23,4	R23=R23 + 4s
addi R13,R13,1	s++
cmpi.eq R13,4,P3,P4	end of for(s=0;s<4;s++)
P4 jmp loc_me_8s_1	
jmp.reg R31	return

Fig. 3.5 An implementation of SAD on the PLX platform

```
Open an implementation;
Tabulate and match memory address with variable names;
Allocate memory spaces for saving the expression of registers;
while (!end_of_program)
    switch (read a instruction)
    {
    case “call (a subroutine)”:
        return address is pushed into stack;
        deal parameters by respective types of parameter passing policies;
        break;
    case “an immediate value or variable name is written to a register”:
        record the value or the name;
        break;
    case “a register is used as a variable”:
        retrieve the name of the variable;
        interpret the mathematic expression;
    case “a register is used as a counter of a control flow”:
        make sure there is an increase/decrease, a comparison and a jump;
    case “jump”:
        jump right before the increase/decrease and comparison (no operation inside)
            $\rightarrow$ a delay loop, output delay time;
        jump somewhere before the register is loaded or after this jump
            $\rightarrow$ a conditional branch;
        jump somewhere after the register is loaded and before the increase/decrease
              and comparison
            $\rightarrow$ a loop,
        The register/memory address used to accumulate or product is extracted;
        // Extract (1) how many terms to be summed/multiplied and (2) array index.
        // Replace array index with proper term number.
        Translate the summation or product in a form of for-loop, $\Sigma$ or $\prod$ respectively;
        break;
    case: “a register is used as an index”:
        load the name or the value pointed by the register;
        break;
    case: “values are wrote to a memory address”:
        output the variable name of the address;
        output the respective mathematical expressions resulting the values;
        break;
    }
// Now the expressions are extracted and part of them may not be good
// for Mathematica to operate.
while (!end_of_extracted_expression)
    if (pre-defined macro in the extracted mathematical expressions)
        translate pre-defined macro into proper Mathematica compliant statements;
if (!Call_Mathematica(extracted_expressions))   // if it is not equivalent with specification,
        Call Error_prediction(extracted_expressions);
```

Fig. 3.6 Symbolic verification and error prediction algorithm

out by setting all the input variables to their maximum and minimum values respectively. To model the error behavior, probability theory and relative axioms are applied.

Let functions $f(x_1, x_2, \ldots, x_i)$ and $f_{DUV}(x_1, x_2, \ldots, x_i)$ respectively be the expected operation without error and the operation carried out by an implementation with possible errors.

Definition 1. Functions $f(x_1, x_2, \ldots, x_i)$ and $f_{DUV}(x_1, x_2, \ldots, x_i)$ are equivalent if and only if:

$$\begin{aligned} &\forall x_i, x_i \in \{\text{the range of the input variables}\} \\ &\Rightarrow f(x_1, x_2, \ldots, x_i) = f_{DUV}(x_1, x_2, \ldots, x_i) \end{aligned}$$

The above equation is the ultimate goal of verification by simulation. Simulation is trustworthy since it authentically reflects the operations carried out by the design implementation. All the combinations of input variable values are subject to check even if "mostly" these all combinations are unachievable. Once every single value in the input variable ranges of functions f_{DUV} and f have been checked and we observe the same result as specified in the design specification or golden model, these two functions are verified to be equivalent.

Lemma 1. *Functions* $f(x_1, x_2, \ldots, x_i)$ *and* $f_{DUV}(x_1, x_2, \ldots, x_i)$ *are equivalent if and only if:*

$$\begin{aligned} &\forall x_i, x_i \in \{\textit{the range of the input variables}\} \\ &\Rightarrow f_{DUV}(x_1, x_2, \ldots, x_i) - f(x_1, x_2, \ldots, x_i) = 0 \end{aligned}$$

Lemma 2. *Obviously, the error e_{DUV} caused by the operations of an implementation is:*

$$\begin{aligned} &\exists x_i, x_i \in \{\textit{the range of the input variables}\} \\ &\quad e_{DUV}(x_1, x_2, \ldots, x_i) \\ &\qquad = f_{DUV}(x_1, x_2, \ldots, x_i) - f(x_1, x_2, \ldots, x_i) \neq 0 \end{aligned}$$

Theorem 1. *If and only if the error, e_{DUV}, caused by the operations of an implementation is 0 for all values in the range of a given input variable, the implementation (design under verification) is equivalent with specification.*

$$\begin{aligned} &\forall x_i, x_i \in \{\textit{the range of the input variables}\} \\ &\quad e_{DUV}(x_1, x_2, \ldots, x_i) = f_{DUV}(x_1, x_2, \ldots, x_i) - f(x_1, x_2, \ldots, x_i) = 0 \end{aligned}$$

or

$$\sum_{x_1, x_2, \ldots, x_i} (e_{DUV}(x_1, x_2, \ldots, x_i))^2 = 0$$

Assume there is at least one error, $e_{DUV}(\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_i) \neq 0$, and the probability of this error $e_{DUV}(\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_i)$ is $p_{(\mathrm{x}_1,\mathrm{x}_2,\ldots,\mathrm{x}_i)}$. Then the total probability of errors of an implementation is: $\mathrm{P}[e_{DUV}] = \sum\limits_{e_{DUV}(\mathrm{x}_1,\mathrm{x}_2,\ldots,x_i)\neq 0} p\,(\mathrm{x}_1, \mathrm{x}_2, \ldots, \mathrm{x}_i)$.

The error caused by ONE combination of different values within its range of an input variable is easy to calculate. Also it is easy to calculate if there are errors for all input value combinations. The reason is that every single value of the input variables/registers can be expressed as one term in a series form. Thus, above computations can be used to check the equivalency and calculate errors discretely as a digital computer does, and the result obtained is mathematically precise without the interferences of discontinuous points.

Here we show how to calculate the probability of errors occurred in a design implementation. Generally the input variables are assumed to have the same uniform distribution over the interval $[0, a-1]$. The constant "a" is 256 for an 8-bit sub-word. Given a derived expression, $\sum_{i=1}^{3}$ Abs[C[i] – R[i]], the result will saturate if it is larger than 255. Thus, its $\mathrm{f_{DUV}}$ is expressed as:

$$\mathrm{f_{DUV}}=\sum_{i=1}^{3}\mathtt{Abs[C[}i\mathtt{]-R[}i\mathtt{]](1-UnitStep[}\sum_{i=1}^{3}\mathtt{Abs[C[}i\mathtt{]-R[}i\mathtt{]]-}$$
$$\mathtt{(a-1)])+(}a\mathtt{-1)UnitStep[}\sum_{i=1}^{3}\mathtt{Abs[C[}i\mathtt{]-R[}i\mathtt{]]-(a-1)],}$$

where "`UnitStep[t]`" is the unit step function. Then the probability of errors occurring is:

$$\mathtt{P[f_{DUV}} > a-1] = \sum_{C[1]=0}^{a-1}\sum_{R[1]=0}^{a-1}\sum_{C[2]=0}^{a-1}\sum_{R[2]=0}^{a-1}\sum_{C[3]=0}^{a-1}\sum_{R[3]=0}^{a-1}$$
$$(\mathtt{UnitStep}[\sum_{i=1}^{3}\mathtt{Abs[C[i]-R[i]]}-(a-1)])/a^6$$
$$= 3a\sum_{i=1}^{a-1}[2(a-i)\times 2\sum_{j=1}^{i}j] + \sum_{k=1}^{a-1}2(a-k)\sum_{i=1}^{a-k-1}[2(a-\mathrm{i})\sum_{j=1}^{\mathrm{i+k}}2j]$$
$$+\sum_{k=1}^{a-1}2(a-k)\sum_{i=a-k}^{a-1}[2(a-i)\sum_{j=1}^{a-1}2j]$$
$$= a(a-1)(a+1)(12-22a+42a^2+43a^3)/(90\mathrm{a}^6) \approx 47.959\%$$

Since Mathematic 4.0 does not handle unit step function perfectly, the summation of unit step function in the above equation is done manually.

3.3 Transaction Level Modeling

Transaction Level Modeling (TLM) is intended for early SoC exploration in the design flow at a relatively lightweight development effort. It is a transaction-based abstraction level residing between the bit-true cycle-accurate model and the untimed algorithmic model. It is performed after function partitioning.

In a digital electronic system, every single component is composed of a finite set of states and a series of concurrent behavior. In TLM notion, components are modeled as modules with a set of concurrent processes that calculate and represent their behavior. These modules exchange communication in the form of transactions through an abstract channel. Depending on the accuracy level required by the corresponding simulation, a channel could be a simple router, an abstract bus model, a network-on-chip model, or some other structure. TLM interfaces are implemented within channels to encapsulate communication protocols. Modules and channels are bound to each other by means of communication ports to establish communication. Once they are bound together, data can be exchanged between them to perform the expected system behavior. A process simply needs to access these interfaces through module ports [29–31].

TLM defines a *transaction* as the data transfer or synchronization between two modules at an instant determined by the hardware/software system specification. The definition of transaction can be refined as a bus-protocol aware structure.

The term *transaction* denotes the set of data being exchanged. A master or initiator is a module that initiates transactions in a system, while a slave or target is a module that receives and serves transactional requests. Any consecutive transactions may have various sizes of data transfer. This variable size corresponds to the amount of data being exchanged between two occurrences of system synchronization. System synchronization is an explicit action between at least two modules that need to coordinate or manage some behavior distributed over them. Such co-operation of different modules is vital to assure the predictable system behavior.

To ensure a proper system functional behavior in TLM SoC simulation, there are two essential points that deserve attention in the modeling process. First, all the data transactions must be blocking, *i.e.*, the thread that initiates the transaction will resume its execution only if the current transaction is completed. Second, all the occurrences of the system synchronization must be potential re-scheduling points in a simulation environment in order to guarantee an accurate simulation of the concurrency. The system synchronization could be modeled by specific means such as event, signal, and interrupt; or by data-exchanges such as polling. If any of these potential system synchronizations causes a call to the simulation kernel, it enables the scheduler to activate other modules. Hence, the simulated system will behave correctly in line with its functional concurrency.

The essence of working out an appropriate model at transactional level lies in the good sense of deciding where and when to implement system synchronization. If too many synchronized points are inserted, the model will tend to be too close to cycle-accurate or RTL models that will not help to gain much simulation speed. Contrarily, if too few synchronized points are implemented, the model may run the risk of having incorrect system execution.

An *untimed communication process* can be generalized as listed in the following steps [31]:

(1) Activate or resume a process.
(2) Read input data for control flow and data processing.
(3) Computation.

(4) Write output data if there is any of them.
(5) Return to Step 2 if more computation is required.
(6) Synchronization:
    (a) if it is "emit-synchronization," return to Step 2;
    (b) if it is "receive-synchronization," the process will be suspended.

System synchronization is very often implemented by an interrupt signal. In the untimed view, an interrupt is however an impulsive system event without any persistence. It is therefore inappropriate to model it using a signal. Instead, a dedicated TLM synchronization protocol with the following features is employed:

(1) Immediate propagation of interrupts from an initiator to a target;
(2) Notice of potential IP internal state change, *i.e.* status register update.

A video decoder decodes 30 frames per second. Sometimes, an untimed TLM inserts functional delay to implement implicit synchronization points for specific timing information. For example, the decoder module can be enabled every 1/30 second by the system timer. From the angle of computational model, such implicit timings bring additional constraints to the execution order of processes in the simulation, and thus reduce the set of possible process interleaves. As a result, the untimed model inserted with functional delay is created as an intermediate level between the purely untimed TLM and the timed TLM.

To develop a timed model at the transactional level, considerations must be given to the time consumption of two aspects: computation and communication.

The *computational delay* is the time amount required to perform specific calculations in characterizing a given system behavior or function; whereas the *communication delay* is the total time consumed in accessing and transferring data or information. The various physical constraints that could bring a significant impact on the system timing behavior such as bus size, bus throughput, or memory size, must also be taken into account during the timed TLM development. *Timed TLM* is able to model in two approaches: annotated model and standalone timed model.

An *annotated model* inserts delay into an untimed model. These annotated delays are the timing information on the micro-architecture level, which make the annotated model distinct from the untimed TLM model inserted with functional delay at architecture level.

A *standalone timed model* denotes a detached model incorporated with the timing information. It is high-level analytical timing models without functional information. They can be built as traffic generators, which model the channel or interconnect traffic with some timing information.

The annotation approach is well suited if the structure of the untimed model already matches the structure of a micro-architectural model, where annotations will be simple wait statements related to the computation time of a specific functionality. The standalone timed model is suitable when the structure of the algorithm is very different from the structure of the micro-architecture.

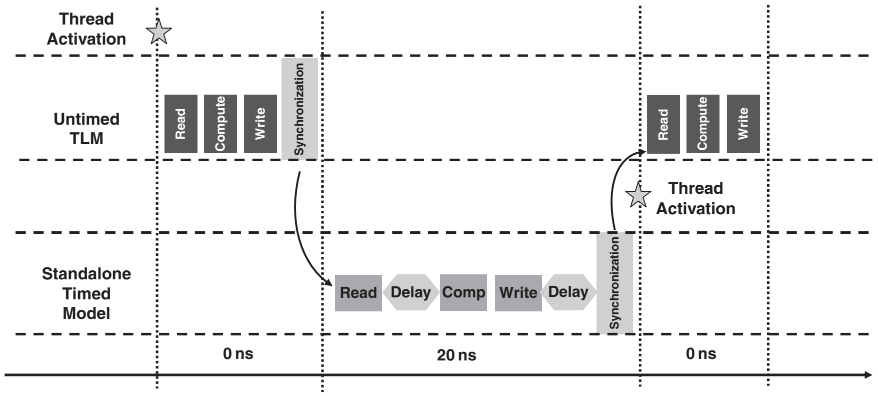


Fig. 3.7 Inter-execution of untimed and timed models

The working concept of the timed TLM can be pictured as an inter-execution of untimed TLM and standalone timed TLM models. Figure 3.7 illustrates the simulation timelines representing the activities of a process execution in the timed TLM [31].

The functional behavior of the untimed model is executed until it reaches a synchronization point. The execution is then passed to the standalone timed model. The timing model will start simulating the delays associated to the functional parts that have just been executed earlier. Meanwhile, time delays of communications and computations are simulated in the timing model as well. Once all of the relevant delays are simulated, the untimed model will resume its execution until the end of its simulation.

3.4 System Level Development Tools

SystemC and SystemVerilog are executable and integrating languages used to complete a platform design from system level to gate level. *StstemVerilog* deals with RTL and below, and *SystemC* deals with those above RTL [32].

SystemVerilog is an extension to the hardware description language Verilog. This was a high-level language specifically oriented to system modeling and verification. It supports linking to externally defined C functions but not the C++ coding styles. Since it was an extension to Verilog, SystemVerilog could naturally handle clock-based modeling without much difficulty. However, it reaches some limitations in the transactional level modeling. The most obvious problem is that SystemVerilog is too close to a hardware-based modeling language. It lacks certain capabilities to handle some aspects of higher-level modeling, for instance, abstract data types are not well supported.

3.4.1 SystemC

In 1999, Coware and Synopsys propose a set of C++ open source class library for hardware modeling. As shown in Fig. 3.8, the essence of *SystemC* lies in the availability of hardware primitives together with a simulation kernel.

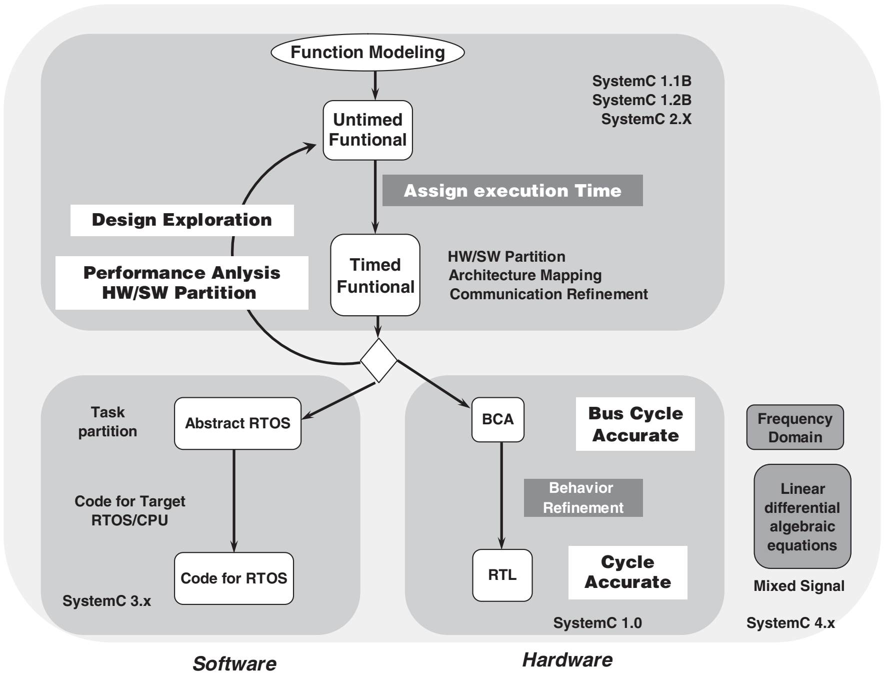


Fig. 3.8 SystemC versions and scopes

With such features, SystemC is able to support multiple abstraction levels and refinement capabilities ranging from high-level functional models to low level timed, cycle-accurate, and RTL models. SystemC holds all of the C++ operator overloading and pointer capabilities. Therefore, software engineers should feel very comfortable to work with SystemC where the job is mostly done in C++. Since it is a C++ based approach, SystemC offers debugging abilities using classical debuggers.

The SystemC class library includes datatypes, cores and channels. *Datatypes* contain logic vectors, bit vectors, arbitrary precision integers, fixed point numbers, C++ built-in types, and user-defined types. *Cores* contain modules, processes, interfaces, ports, events and event-driven simulation. Elementary *channels* contain signal, timer, mutex, semaphore, FIFO, *etc.*

SystemC provides a number of datatypes that are useful for hardware design. These datatypes are implemented in C++ classes. The `sc_int` and `sc_uint` support for finite precision signed/unsigned signals; `sc_fixed` and `sc_ufixed` support for fixed-point signals; `sc_logic` supports for 4-level logic values (0, 1, x and z).

A SystemC module represents an individual identifiable hardware element. A *module* definition defines the port-level interface of a module, its internal storage elements and its behavior. The synthesizable subset supports modules declared as a `class` or as a `struct`. In addition, specialization of modules using templates is supported. The module definition can use either the `SC_MODULE` macro or a derivation from `sc_module` class. A module member contains signals, sub-modules, constructors and processes.

A *module constructor* is declared by `SC_CTOR` macro. Submodule instantiations, port mappings, and process statements are located in the module constructor.

Ports represent the externally visible interfaces to a module and are used to transfer data into and out of the module.

A *process* is declared by `SC_METHOD` or `SC_CTHREAD`. The `SC_METHOD` must not contain any wait statement or any invocation of a function which may directly or indirectly cause the execution of a wait statement. Consequently, it must not contain any loop which is not unrollable. A *method* is triggered by a signal switching event, denoted in a sensitive list. The `SC_CTHREAD` must be an infinite loop to prevent execution reaching the end of the process. Each unbounded loop must contain at least one explicit `wait()` in each control path.

Following shows a simple SystemC example:

```
SC_MODULE(ALU)
{//ports
 sc_in_clk           clk;
 sc_in<sc_uint<1> >  op;
 sc_in<sc_uint<32> >  a, b;
 sc_out<sc_uint<32> > z;
   //internal signals
   sc_signal<sc_unit<32> > z1, z2;
   //sub-modules
   ADDER *func1;
   MULTIPLIER *func2;
    //process
   void execute() { z.write(op.read()?z1:z2); }
   //construct

SC_CTOR(ALU)
 {
        func1=new ADDER("func1");
        func1->a(a); func1->b(b); func1->z(z1);
        func2=new MULTIPLIER("func2");
        func2->a(a); func2->b(b); func2->z(z2);
     SC_METHOD(execute);
     sensitive_pos<<clk;
    }
}
```

3.4.2 LISA

The *Language for Instruction Set Architectures* (LISA) is an *architecture description language* (ADL) specific for embedded processor development. LISA has been developed at the Institute for Integrated Signal Processing Systems at the RWTH Aachen University and is commercialized by Coware Inc [33–36].

At architecture level, a design process contains the following 4 tasks:

(1) *Architecture Exploration*: Select an architecture for the target application. This task is composed of three main works. First is hardware/software partitioning. The critical portions of the application that require hardware support is determined by profiling. Second, decide the embedded processor instruction-set by the partitioning and profiling results. Third, fine-tune the micro-architecture of the processor. This phase is an iterative optimization process which is repeated until a sufficient fit between the selected architecture and the application is obtained.
(2) *Architecture Implementation*: Implement the processor architecture description into RTL.
(3) *Software Application Design*: An Instruction Set Simulator (ISS) and a compiler are developed.
(4) *System Integration and Verification*: An approximate-timed or cycle-accurate hardware/software co-simulation is required.

In ASIP design, the key to success is to have sufficient exploration of the architectural design space. The LISA language allows engineers to implement changes to the architecture model quickly, as the level of abstraction is higher than RTL. The LISA model of the target architecture is used to automatically generate software tools such as instruction encoding generator, C-compiler, assembler, linker, simulator and profiler. These software tools are used to identify hot spots and to jointly profile the architecture and the application. Both are optimized according to the profiling results, *e.g.*, throughput, clock cycles or execution count of instructions. This exploration loop can be repeated until the design goals are met. With LISA, a highly efficient design space exploration is ensured, as the ASIP model can be modified easily and the software tools are re-generated within a negligible amount of time.

The knowledge about the physical characteristics of the architecture is important at an early design stage already. For example, the information about clock speed substantially influences the number of pipeline stages or even the pipeline organization in general. Ignoring physical parameters in the design space exploration phase leads to suboptimal solutions or long redesign cycles. The automatic ASIP implementation from LISA provides important feedback about the physical characteristics of the architecture.

In LISA, the instruction-set is defined by a directed acyclic graph. Information about the exclusiveness of operations is provided inherently by the graph structure.

A LISA model basically consists of two parts: the specification of the resources of the target architecture and the description of the instruction-set, behavior, and timing. The first one is captured in the resource section, while the latter one is described by the LISA operation graph.

The *resource section* defines the resources of the ASIP such as storage elements, functional units, and pipelines. All storage elements are declared with a global scope by specifying a data-type, identifier and optionally a resource-type. The memory is specified by the keyword `RAM`. The size is defined by the number of blocks the memory consists of and the bit size of each block. The pipeline definition comprises the pipeline name, an ordered list of pipeline stage names and the pipeline register elements between pipeline stages. Functional units are declared by the keyword `UNIT` followed by a unique name and a set of LISA operations. The RTL hardware model structure can be derived explicitly from the specification of the pipeline and implicitly from the global scope of the defined resources. The following code shows a LISA resource section example.

```
RESOURCE {
   RAM int datamem {
     SIZE(65536);
     BLOCKSIZE(32,32);
     FLAGS(R|W);
   };
   PROGRAM_COUNTER unsigned short PC;
   REGISTER int RegF[0..31];
   PIN IN int ins;
   PIN OUT int outs;
   PIPELINE pipe = {FE; DE; IS; OP; EX; WB};
   PIPELINE_REGISTER IN pipe {int op1, op2, result;};
   UNIT Fetch {fetch;};
   UNIT Bypass {bypass_fifo;};
   UNIT MemAccess {data_mem_read;};
   UNIT Alu {ADDI, SUBI, MOVI, MOVM, CMPI;};
   UNIT WriteBack {writereg;};
}
```

The instruction-set architecture is defined by a directed acyclic, *operation graph*. The nodes of this graph are LISA operations. Each node may contain information about the assembly syntax, the binary encoding of the instruction-set, the behavior and timing of the architecture. Generally, this information is specified independently by coding section, syntax section, behavior section, and activation section.

The LISA *coding section* is utilized to generate the corresponding decode logic. The bit pattern of a terminal coding element can be compared directly with the instruction word. The bit patterns represented by a non-terminal coding element are derived from the corresponding sub-graph.

The *syntax section* covers the assembly syntax of the instruction-set and is comparable to the coding section. The non-terminal syntax elements represent the mnemonics of the instruction-set.

The *behavior section* covers the state update functions of the architecture. They are described using C-programming language. Following code shows an example of behavior description.

```
OPERATION load_operand {
 BEHAVIOR{
   opreg = instr.Extract(18,10);
   if (opreg == last_resultreg) operand = alu_result;
   else operand = RegF[opreg];
   pipe_reg = operand;
 }
}
```

The *activation section* describes the temporal relation between LISA operations. It is a set of activation elements used to schedule the operations of an instruction for execution. By the scheduling spatial and temporal relation, the pipelining micro-architecture of the instruction-set can be explored, and the first hardware representation obtained. During exploration, the exclusiveness of resource conflict is analyzed. For example, if the program-counter increase operation and memory-address generation operation in a memory load instruction do not use the same control signal, the two operations can be put in the same pipeline stage.

LISA uses behavioral synthesis to map the architecture description into RTL. The process is shown in Fig. 3.9.

The *structuring process* maps the entities in a resource section into registers, memories or pipes. The *functional-mapping* process maps functional units into an HDL process, such as Verilog `always-block` or SystemC `method`. The *interconnect-mapping process* converts paths into signals, including multiplexer and wires.

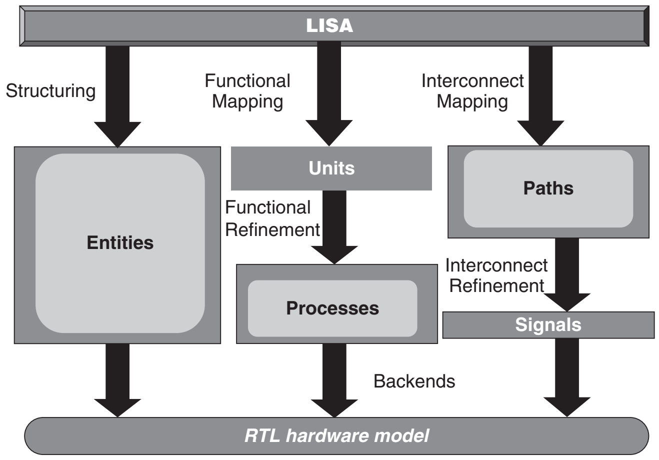


Fig. 3.9 LISA behavior synthesis process

On mapping into RTL, optimization of area, time and energy is what a designer concerns on. The optimization can be obtained from the explicit architectural information specified in the LISA model. Each kind of optimization requires an appropriate abstraction level and suitable data-structures to retain the architectural information.

The behavior description can be represented by CFG and DFG. The first simplification, which removes unnecessary multiplexers, duplicated operators, and constant inputs, is performed on DFG by methods of constant propagation, multiplexer simplification, structural simplification and merging identical nodes.

The resource sharing optimization reduces the chip area but increases critical path delay, thus it is a trade-off by constraints. An example is sharing addition and subtraction operations on a single adder by insert 2's complement logic and multiplexer on input port. At abstraction level, the physical information is unknown. LISA uses virtual gate to estimate area and timing. For example it assumes that adder uses carry-propagation and multiplier uses Booth model.

The problem of finding the minimum number of required hardware resources can be solved with the help of compatibility graphs and conflict graphs. Using a *compatibility graph*, all nodes of a fully connected subgraph called clique can be mapped on a single resource. Instead, if using a *conflict graph*, all nodes of a completely unconnected subgraph can be shared. To achieve minimal resource usage, the minimal number of such subgraphs has to be found. In graph theory, this problem is known as the clique covering problem for compatibility graphs and as the graph coloring problem for conflict graphs. The graph coloring problem stems from the goal of using the minimum number of colors for coloring the different countries on a map without using the same color for neighboring countries. Translated to conflict graphs, the vertices represent the different countries while edges connect pairs of neighboring countries.

After the instruction-set decoder is generated, LISA toolset can be used to generate some software tools, including C-compiler, assembler and simulator.

LISATek relies on CoSy Express which is derived from the CoSy compiler development system from Associated Compiler Experts (ACE) [37]. CoSy is the professional, easy-targetable and highly flexible compiler development system in creating high-quality, high-performance compilers for a broad spectrum of microcontrollers to CISC, RISC, DSP and VLIW processor architectures.

CoSy's modular design, surrounding a generic and extensible intermediate representation (IR), offers numerous configuration possibilities both at the IR level and the backend for machine code generation. CoSy generates compilers from so-called Code Generator Description (CGD) files. A CGD model consists mainly of the following three components:

(1) Available target processor resources like registers or functional units.
(2) A description of mapping rules, specifying how C/C++ language constructs are mapped to assembly instructions.
(3) A scheduler table describing instruction latencies and resource usage.

Apart from that, some further information like function calling conventions, C data type sizes, and alignment are required. The CGD description is generated from the LISA description model.

To handle the enormous complexity, system level simulation is absolutely necessary for both performance evaluation as well as verification in the system context. The earlier design errors or performance shortcomings are detected in the design flow, the lower the cost for redesign cycles becomes. The automatically generated LISA processor simulators can be integrated into various system simulation environments, such as CoWare convergenSC or Synopsys CoCentric System studio. Thus, modules provided by different design teams or even third parties can be combined easily.

Verification is an essential part of any processor design. Moving from one abstraction level down to another one the designer is required to add additional implementation details for optimizations. LISATek approach reduces the verification effort by automating large parts of the design flow from a high-level functional processor model down to the lower level of a synthesizable hardware implementation.

Chapter 4
Embedded Processor Design

Today's growth in markets for consumer electronics, wireless electronics, and handheld computing requires cost-efficient solutions that supply high performance computing, energy efficiency, and programmability. General-purpose processors are poorly suited to meet the requirements of energy efficiency and competitive cost. ASICs are unable to provide sufficient programmability. As the result, a variety of Application Specific Instruction-set Processors (ASIP) is emerging to meet the requirement.

To compete with ASIC, an ASIP requires better performance/power efficiency than a general purpose processor. To achieve this target, an ASIP must require more parallelism in various levels, such as data level parallelism (DLP), instruction level parallelism (ILP), and thread level parallelism (TLP) that will be depicted in this chapter.

4.1 Specific Instruction-Set

The earliest specific instruction-set example is floating-point. Floating-point operation hardware is much more complex than integer operation. Early designed processors can only do integer operations, and implement floating-point operation by software emulation. Most scientific algorithms require floating-point, but emulation implementation is too slow for them. Thus, scientific requirement drives processor to integrate floating-point operations.

Most multimedia applications use fixed-point operations. Due to the limitations of human eye and ear sensitivity, some precision loss on image pixels and audio samples is acceptable. For example in a DCT algorithm, using 12-bit fixed-point to represent a cosine value is good enough for most image quality requirements. Fixed-point operation can be simply an integer arithmetic operation and a shift operation. In typical integer addition and multiplication operations, since the most-significant bits (MSB) are truncated when the result has an overflow, the following shift operation will get a wrong value. Thus for multimedia applications, the fixed-point operation applied should be able to preserve MSB.

Multiply-Accumulate (MAC) is a key operation in Finite Impulse Response (FIR) filter function. Some DSP processors implement MAC with automatic looping and index increase, which can process $\Sigma(a[i] \times b[i])$ as a single operation. Since MAC is composed of multiplication and addition operations, it is always the longest path in an ALU. In a RISC architecture that requires all instructions to be executed in one cycle, MAC becomes the bottleneck. Recent design utilizes VLIW wide-issue capability to implement MAC and remove the long critical path.

Division operation requires subtraction and shift operations for each divider bit, thus the path delay for a 32-bit division is extremely high. Many general purpose processors use floating-point unit to compute integer division. Rare fixed-point embedded processor supports division operation. In that case, a compiler is used to convert division into a loop with subtraction and shift operations to reduce hardware cost.

Saturation arithmetic is useful for multimedia applications. When two image pixels or audio samples are mixed, their intensions are added. By typical integer addition, mixed white pixel will become light gray when its MSB is truncated. To avoid the wrong result, software should check all pixels and keep the mixed intension as a maximum white value when overflow occurs, which is very heavy work. The saturation arithmetic instructions are thus implemented in hardware to saturate the overflow/underflow result to an upper/lower bound to reduce the error.

Permutation operations are helpful in many algorithms. Datatype conversion is the basic permutation operation in all processor. Reverse and butterfly ordering is widely used in FIR and FFT. Many symmetric-key cryptographic algorithms such as DES and AES are based on complex permutation. The selection of permutation instructions to implement is very different between embedded processors.

4.2 Data Level Parallelism

Vector supercomputer was developed in the 1960s to increase the scientific computation speed. Since scientific program codes contain many one-dimensional vector and two-dimensional matrix operations, using a vector processor can perform these operations simultaneously to improve performance. A vector processor is also called a single-instruction multiple-data (SIMD) machine because it can apply one instruction on many data elements. Such kind of parallelism is often called *data level parallelism* (DLP).

4.2.1 SIMD

Two main vector processing techniques will be introduced in this section. One uses processor array; ILLIAC-IV [38] is a representative example; Another uses automatic looping; a good example is Cray-1 [39].

ILLIAC-IV has 256 processing elements (PEs), which are partitioned into four groups; each group contains sixty-four PEs and one control unit (CU). The sixty-four PEs are structured as an 8×8 array as shown in Fig. 4.1. Each PE can communicate

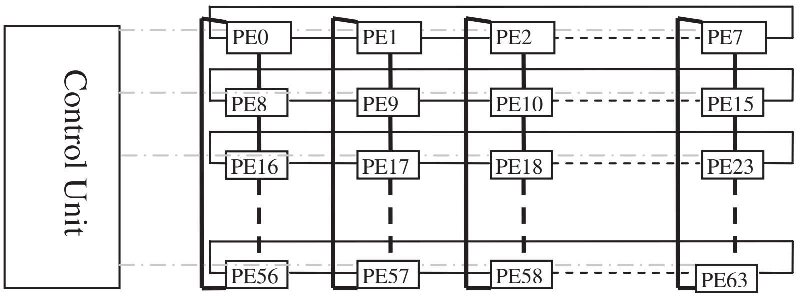


Fig. 4.1 ILLIAC-IV architecture

to its four neighbors by a data routing network. A PE contains a 64-bit ALU and a 2048-word memory, each PE can only access its own local memory. CU decodes instructions and executes conditional test and branch instructions, and can access the whole memory array.

Assume that we want to run the following code:

```
for(I=1;I<=64;I=I+1) C[I]=A[I]+B[I];
```

To parallel process the code, data in memory should be distributed over PEs, such that each PE owns one data element. ILLIAC-IV uses 3 instructions to perform this code: the first instruction parallel loads data elements of *A* into PE accumulators, the second instruction adds the data elements of *B* to the accumulators, and the third instruction stores the contents of each accumulator into memory *C*. Arrangement of data in memory becomes a primary consideration for efficiency. If array *A* is allocated in a single PE, the loop should process sequentially by memory access limitation. It occurs on a two-dimension array in which row dimension is distributed over PEs, but the operation works on the column dimension.

Cray-1 has independent scalar and vector function units. As shown in Fig. 4.2, eight vector registers are used for vector operations, and memory (data) elements are loaded into vector registers before execution. Each vector register has sixty-fourwords. The vector function unit is a 64-bit integer ALU with a smart data forward path.

Using Cray-1, the above-mentioned sample code can be translated into two vector memory load instructions: one vector addition instruction and one vector store instruction. Data elements of *A* are loaded from memory into vector register word by word. When the first element of *B* is loaded, it is forwarded to the vector function unit accompanied with the first element of *A* from the vector register and the vector addition can start execution without waiting *B* fully loaded. That is, memory load and addition operations work in pipeline. The vector function unit continuously generates sixty-four addition results in sixty-four cycles. While it processes word by word, data in memory do not need special organization, and the performance for row vector and column vector is the same, which is a benefit

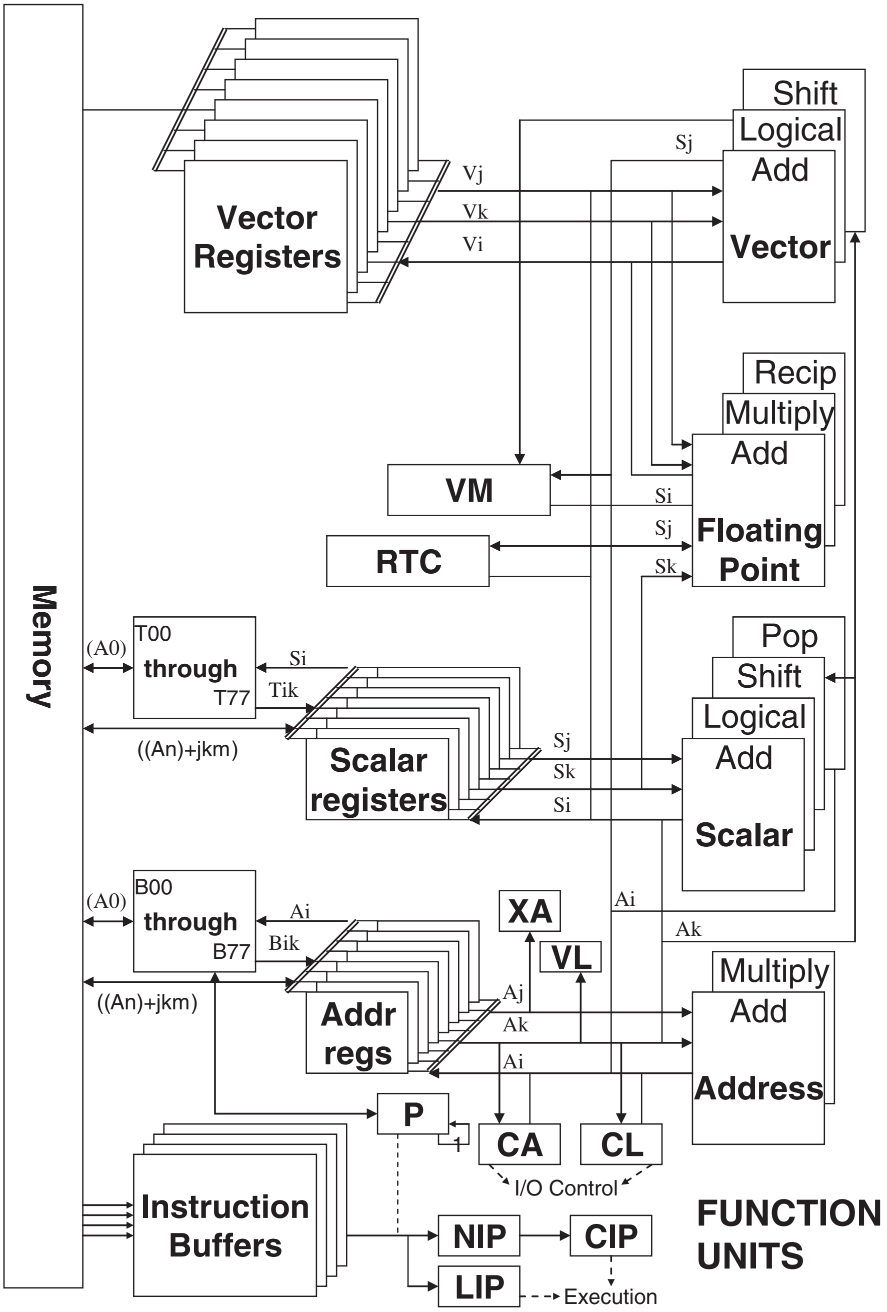


Fig. 4.2 Cray-1 architecture

compared to the processor array architecture. Compared to a scalar processor, since each instruction is only fetched and decoded once, the sixty-four loop iterations can be automatically performed by the vector function unit, which saves the branch overhead, the main waste on processor execution. Thus, the efficiency of MOPS/Watt of a vector processor is much better than a scalar processor.

4.2.2 SWP-SIMD

Multimedia applications mostly perform low-precision data, such as 16-bit audio samples and 8-bit video pixels. Today the ALU word size in a processor is mostly sixty-four bits. It is a waste to compute 16-bit data using a 64-bit ALU. If the 64-bit ALU can compute four 16-bit data simultaneously, its throughput can be higher. Instruction set architecture (ISA) with this feature is called a *subword-parallel single instruction multiple data* (SWP-SIMD) processor [2]. It works as an SIMD vector machine, but performs in a single register. Many low-precision data are packed into a *superword* which occupies a register, and each element is called *subword* which only occupies part of a register. This feature is also called *multimedia extension* for it is specified for multimedia applications. MAX-1 is the first SWP-SIMD ISA for HP PA-RISC processor [40], introduced in January 1994. Other famous examples are MMX/SSE for Intel IA-32/64 [41, 42], VMX/AltiVec for IBM PowerPC [43], 3DNow! for AMD K6 [44], VIS for SUN SPARC [45], and MDMX for MIPS processors [46].

PLX [1] is an SWP-SIMD ISA developed by Professor Ruby Lee at Princeton University. The main feature of PLX is that it is native to SWP-SIMD. Its vector function unit supports 8/16/32/64 subword widths, and its scalar function unit is just the 64-bit subword subset in the vector unit. A typical multimedia code contains many scalar operations, such as loop counter or memory index, which disable vector pipeline to execute smoothly. A native SWP-SIMD can execute scalar and vector operations in the same core to reduce scalar-vector communication overhead.

Power-aware is a benefit obtained from the SWP-SIMD feature. The term power-aware is often ascribed to any system which design has been sensitive to energy consideration; its connotation in recent work has been shown in [47]:

(1) The system allows its clients to adjust the expected quality and also the tolerable latency/throughput constraints.
(2) When such adjustments are made, the energy consumption is expected to vary accordingly *i.e.*, higher energy dissipation is tolerated by clients for higher quality (or lower latency) and vice-versa.

There are many topics to work on power/performance trade-offs. On the circuit level, since the CMOS power consumption is proportional to voltage square, the core and bus buffer supply voltages usually have to be reduced to save power. On the logic level, gated clock when datapath is not working can reduce unnecessary logic switching power. On the system level, the supply power of a non-active core can be turned off. The disadvantage is its requiring a long stable time to turn on again, which may cause real-time request failure.

On the algorithm level, datapath width adjustment can get the most power budget. For example, if a program performing only 8-bit operations with a value range of -128 to $+127$ is implemented in a 32-bit ALU, the register switching of bits 8 to 31 are meaningless and the power is thus wasted. Most applications contain variables of different widths. An MPEG-2 video decoder [48], for example, contains fifty 1-bit Boolean variables, nine 8-bit char variables, thirty-nine 16-bits short variables, seventeen 24-bit variables, and eighty-two 32-bit variables. If implemented in a 16-bit datapath, the 24-bit and 32-bit operations cannot be completed in one cycle and the performance will be degraded, but the power spent on fewer bit operations is saved by the reduction of meaningless switching. This example showed that when the datapath width is larger than 28 bits, the performance increases little, but the power and area are still increased linearly, so the best power-efficient design occurs at 28 bits.

Most processor-based system design is unable to change the datapath width, or they need to change instruction set architecture to adjust datapath width [49], which needs extra cost for decoding the second instruction set. PLX's native subword-parallelism design extends the flexibility to change datapath width during software execution, which can improve the computation power efficiency.

Figure 4.3 demonstrates the subword parallel processing concept. Eight 8-bit data are packed into one 64-bit word. They are processed by one `padd` instruction, taking only one cycle. With the appropriate subword boundaries, this technique results in the parallel processing of subwords. The degree of parallelism is within an instruction and depends upon the size of the subword.

Figure 4.4 shows a logic level power-aware concept. Figure 4.4(a) is a 4-bit Wallace-tree pipeline adder. The maximum delay (T) is two half-adder delays at stage 4. The highest performance is 1/T operations per second. When the system requires only half of the performance, the clock frequency can reduce to 1/2T, and the power consumption is also reduced to half. Now the clock cycle 2T is much larger than the maximum delay, T. Figure 4.4(b) changes the pipeline registers of Stage 1 and Stage 3 into buffers, the critical path delay is one full-adder plus two half-adders plus register setup time, it is a little lower than 2T. The combinational logic propagation power is increased because it is more complex, but register power

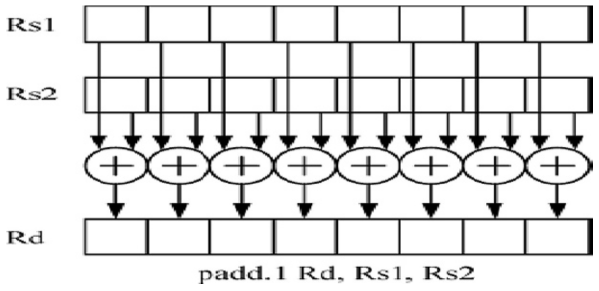


Fig. 4.3 Subword-parallel execution

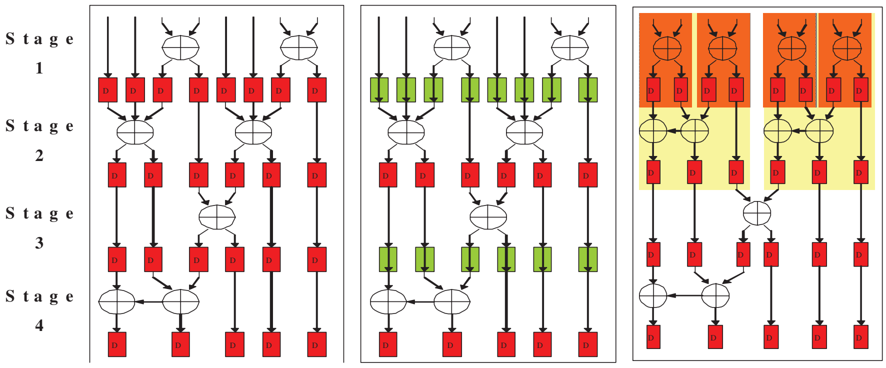


Fig. 4.4 Power-aware reconfigurable pipeline adder

is reduced. Using well-designed combinational logic, the total power consumption can be reduced greatly. Figure 4.4(c) extends the adder to support subword parallel. Compared to Fig. 4.4(a), it uses 4 extra adders, but can compute four 1-bit additions in one stage, two 2-bit additions in two stages, or one 4-bit addition in 4 stages. When data precision is low, higher stages can be gated to save power.

Adjusting the pipeline structure dynamically will increase the complexity of data dependence detection. An instruction containing read-after-write (RAW) dependence is caused by the use of a previous instruction result in one of the operands. In a pipeline structure, the previous result is not written into a register file when this instruction is executing. To solve this kind of dependence, the result has to directly be forwarded to the ALU operand port. On a dynamic pipeline architecture, the result may have to be forwarded from any of the four other previous stages to avoid RAW conflict, which increases the pipeline control complexity.

Another subword-parallel ALU design is for high-performance purposes. Figure 4.5 shows a 64-bit wide carry-select adder structure, where all the subword 8-bit adders are designed to complete an addition in one clock cycle. At the beginning, two pairs of 8-bit subword additions are computed in each 8-bit ALU,

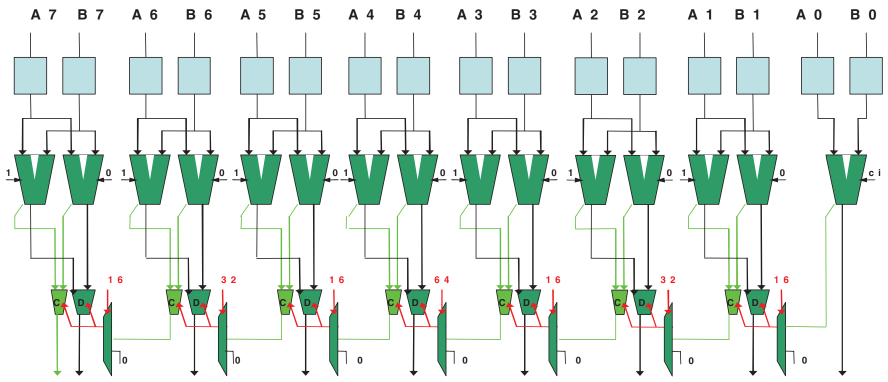


Fig. 4.5 High-performance subword-parallel design

one with a carry-in of 0 and the other with a carry-in of 1. Then these two obtained results are respectively stored in the two registers waiting for the select control signal to select an addition result to output. In such way, we can have eight 8-bit precision addition operations done in one cycle. For a 16-bit precision addition, the four multiplexer control signals "16" are high and the other "32" and "64" control signals are low, such that the carry-out of an even byte can pass through the multiplexer and serve as the select control signal to select the result of an odd byte, we can thus have four 16-bit subword addition results generated at one clock cycle. For a 64-bit precision addition, all the multiplexer control signals are high, and we can have one 64-bit full-bword addition result generated, which datapath delay is the longest, equal to the delay of one 8-bit adder plus those of the fourteen multiplexers. In such design, all instructions are required to be completed in one cycle, making the pipeline control simpler.

No register on this architecture is able to turn off, but we can reduce power supply for lower data precision. When no 64-bit data exist in the whole application, the actual critical path delay is the 8-bit adder delay plus six multiplexer delays, which is about 80% of the 64-bit datapath. By CMOS theory, the supply voltage can be reduced to get a 36% power reduction.

4.3 Instruction Level Parallelism

The SIMD and SWP-SIMD processors perform a large amount of data in one cycle. But not all algorithms contain data level parallelism. To improve performance, a processor should be able to execute many instructions in one cycle, which is called *instruction level parallelism* (ILP).

4.3.1 SuperScalar

Cray's CDC 6600, built in 1965, is the first *superscalar processor* to achieve *instruction level parallelism* (ILP) by dispatching straightforward instructions into multiple functional units simultaneously and executing them in one cycle.

A *scalar ALU* contains many arithmetic components such as adder, multiplier and Boolean logics. But only one component is able to work at a time. An ILP processor allows many components to work in parallel and sometimes duplicate ALUs with often used components to increase parallelism. An issue-logic is used to dispatch instructions into ALUs. The actual parallelism is limited by operation dependence. Two dependent operations cannot be issued simultaneously.

Parallelism can be improved by two techniques. Dependence removal removes false dependences by variable rename and scalar expansion. Rescheduling moves independent operations out of dependent operations to have more operations executed in parallel. The rescheduling method is also called out-of-order execution, because the operation execution order is different to that in the given code.

Superscalar processor implements the above two techniques by hardware. Performing dependence removal needs a large renaming buffer. Performing out-of-order execution needs an efficient instruction fetch buffer to get enough operations to issue, and a retire logic to remove the executed operations.

Intel Pentium 4 [50] is a *superscalar processor*, which micro-architecture is shown in Fig. 4.6. Its CISC instructions are decoded into RISC-like micro-operations (μop). The low latency trace cache can deliver up to three μops per cycle for out-of-order execution. The three ALUs can work in parallel. Two of them can only process simple operations but have double throughput. The 128-entry renaming buffer works as a register file for RISC-like μops. The register renamer maps the eight logical IA32 registers such as EAX onto the 128 physical registers.

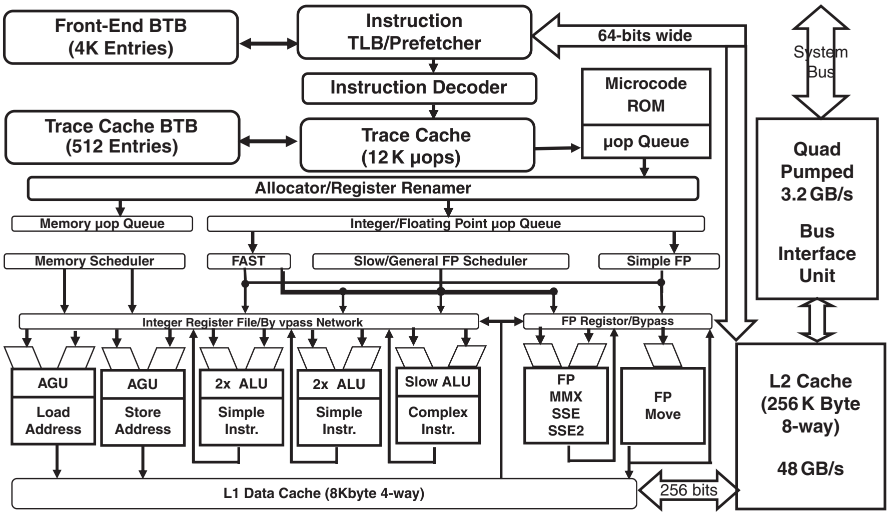


Fig. 4.6 Pentium 4 processor micro-architecture

4.3.2 VLIW

Very Long Instruction Word (VLIW) processor implements dependence removal and operation rescheduling by a compiler. The hardware cost of implementing these two techniques in a superscalar processor is high; it is not affordable for portable devices. Since operation dependences can be determined in a program code, it can be optimized by a compiler to save hardware cost. The disadvantage is that software needs re-compilation when processor micro-architecture is changed. It is not accessible for general purpose desktop processor; it is only feasible for embedded processor. A compiled instruction contains many operations like a long horizontal microcode.

Typically a VLIW instruction is a pack of many scalar instructions, which brings its name VLIW.

TI TMS320C6 DSP is a VLIW processor [51]. It contains 8 execution units and is partitioned into two clusters. Each cluster contains four execution units named as M, L, S, and D. Multiplication operations can only be executed in the M units. Logic, shift and memory operations can only be executed in the L, S, or D unit. Arithmetic operations can be executed in the L, S, and D units. At most 8 operations can be executed in parallel.

Figure 4.7 shows a generic *VLIW datapath* architecture. In designing VLIW processor, the first challenge is fan-out. When 8 ALUs are working, each ALU needs to get two operands from a register file and write one result into the register file, thus the register file requires 16 read ports and 8 write ports. Figure 4.8 shows a register file structure. When the number of access ports doubled, the routing area is squared. That is, the register file will dominate chip cost in both area and speed, which is not

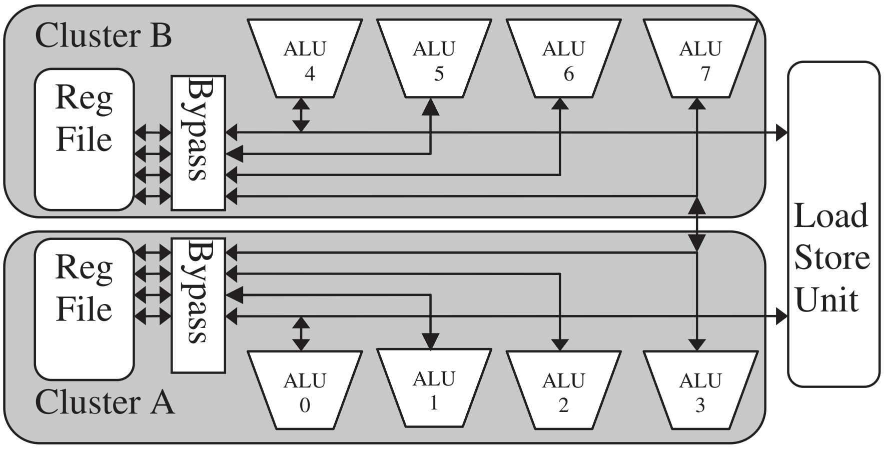


Fig. 4.7 A generic two-cluster VLIW architecture

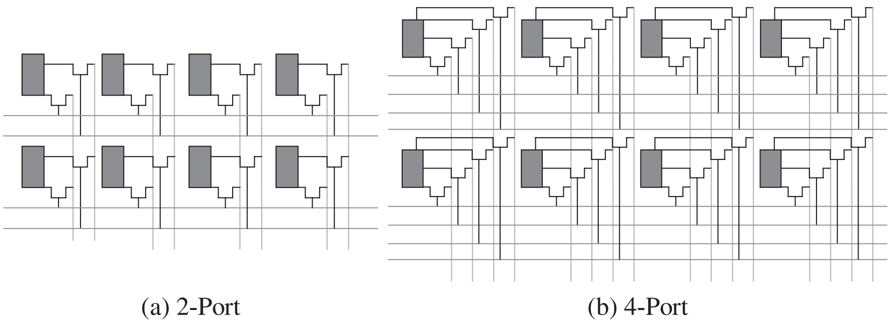


Fig. 4.8 Register file

what we want. This problem is similar to the load/store unit design, where allowing many ALUs to access memory will increase cache complexity.

The *bypass logic* is another problem. On a pipelining RISC architecture, ALU result is buffered in a temporal result register before it is written into the register file. To avoid blocking on continuous read-after-write dependent instructions, a bypass logic is used to forward ALU result in a previous instruction to the ALU input port or operand register in a current instruction. Figure 4.9 shows the bypass logic on a 2-issue VLIW processor. Each ALU result needs to be forwarded to all ALU input ports and operand registers, thus the eight ALUs need thirty-two paths. A large fan-out induces a long wire delay.

To reduce the fan-out problem, a many-issue VLIW processor is usually clustered as shown in Fig. 4.7, where a register file and ALU are partitioned into clusters. The communication between clusters is implemented on special instructions that perform on a specific ALU. Only one load/store access is allowed to a cluster to reduce memory complexity.

Control flow handling is more important for VLIW processor design. A control flow induces a conditional branch, which may cause instruction stream change. On a VLIW processor, when two ALUs generate different branches, which will be the next instruction to execute?

A simple way is to avoid packing multiple branch operations in one cycle, but it will degrade performance. The popular solution is using predication execution, or so-called if-conversion. This technique changes control flow into data flow by introducing a condition expression to be the third operand of the operation. Then the instruction stream can be in one line, which will be described in Section 5.1.4 in more detail. The implementation of predication requires extra flags to store the comparison result, which will be passed to ALU as the third operand.

On a pipelining architecture, branch induces pipeline re-fill that wastes computation power. On a VLIW processor, a pipeline stage contains many operations, thus the waste becomes higher. A solution is using unbundled branch technique.

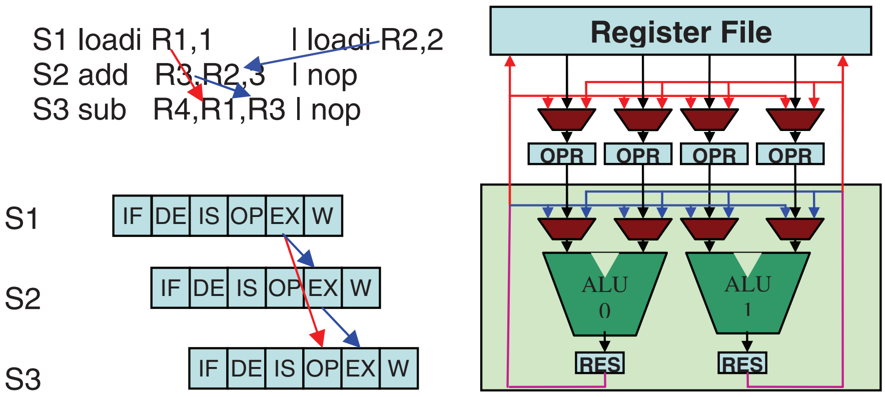


Fig. 4.9 Bypass path in a 2-issue VLIW

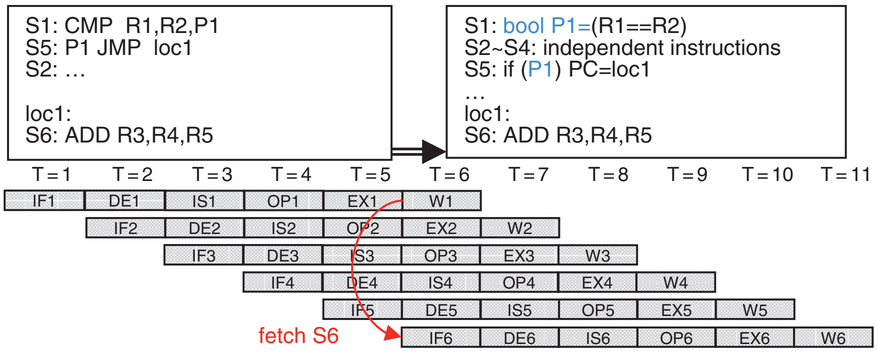


Fig. 4.10 Unbundled branch

Figure 4.10 shows this technique. Figure 4.10(a) is a traditional RISC code. After the branch instruction S5 executed, the pipeline has to be cleared and refilled by S6. Figure 4.10(b) is a code using unbundled branch. At T = 5, when the comparison operation in S1 is executed, if the register P1 is set, the code will jump to the target, where S6 is just beginning its instruction fetch stage (IF), but note that many other pipeline stages in S2–S4 are not cleared. If they are independent instructions, S2∼S4 in the pipeline can continue their executions without wasting time. Then, when S5 reaches T = 9, the program counter (PC) is set for S6 execution. That is, unbundled branch technique needs to insert independent instructions by the compiler and the number of instructions inserted depends on the depth of the pipeline stages.

4.3.3 NISC

No Instruction Set Computing (NISC) processors [52] are the successor of VLIW processors with a much simpler hardware. In contrast to the VLIW instruction words being mapped to the microcode level, the NISC instruction words are directly mapped to the datapath control words. It provides more flexibility and opportunities for both horizontal and vertical controls of the operations in the processor datapath. The NISC instruction word size is larger than VLIW, which does not favor off-chip memory bandwidth, but is feasible for SoC.

Complex Instruction Set Computing (CISC) was necessary in an early age. Since the memory capacity was small at that time, designers tried to improve code density by constructing complex instructions doing compound functions at one time. Each complex instruction took several clock cycles, and datapath control words for each clock cycle were stored in a much faster micro program memory (μPM). The concept of micro programming allowed emulation of any instruction set and construction of specialized instructions, while speeding up the execution. An old 8086 code can be executed on a Pentium 4 processor as shown in Fig. 4.6 to get a 1000X speedup without any modification.

The design philosophy of *Reduced Instruction Set Computing* (RISC) emerged in the early 1980s [53]. All instructions in a RISC are simple and can be executed in one clock cycle. This allows datapath to be efficiently pipelined. The μPM was replaced with a decoding stage that follows the instruction fetch from program memory (PM). Since instructions are simpler, a RISC needs more code for the same work and the size of the program memory is larger.

RISC is defined as a processor with most of these characteristics [54]:

(1) Instructions are conceptually simple;
(2) Instructions are of uniform length;
(3) Instructions use one (or very few) instruction format(s);
(4) Instructions use one (or very few) addressing mode(s);
(5) The instruction set is orthogonal;
(6) The architecture is a load-and-store architecture;
(7) All instructions are register-to-register operations;
(8) Almost all instructions execute in one clock cycle;
(9) It has a large number of general-purpose registers; and
(10) The architecture has hard-wired control unit.

The idea of utilizing RISC technology for a CISC processor is brought from the Intel P6 processor project at early 1990s while the debate of RISC/CISC was still boiling. Most RISC technology benefits can be imported into a CISC design by using the simplified philosophy listed as follows: (1) build hardware in which almost all of the operations are simple and fundamental operations, (2) operands are simple data kept in registers or fetched from/to memory by the load/store operations. Intel breaks the lengthy CISC instructions into simpler micro-operations that more closely resemble RISC instructions. The micro-operations are then fed into a core that takes advantage of the latest RISC innovations.

Both CISC and RISC spend more logics to maintain incoming instruction stream, especially the branch prediction. A branch operation needs to refill pipeline stages before execution unit. A general code contains 1/7 branch operation. Thus, deep pipeline architecture will waste much time on branching. To achieve higher clock speed involves pipelining the micro-architecture to finer granularity. Since a branch limits the performance of pipelining, it forces more powerful branch prediction development. It is the main reason that the power-performance efficiency of a software implementation is lower than the specific hardware implementation.

The NISC instruction word is directly sent to the datapath which does not own any pipeline stage, thus its branch overhead is ignorable. With its simplified processor architecture, a NISC datapath can be deeply pipelined and duplicated to get vertical and horizontal parallelism. The components on the datapath not only consist of computation units, but also registers and memories. It allows NISC to process very complex operations in a cycle. Most of the power in a NISC is spent on computation in the datapath, thus its power-performance efficiency is close to a specific hardware implementation, but keeping better software flexibility (Fig. 4.11).

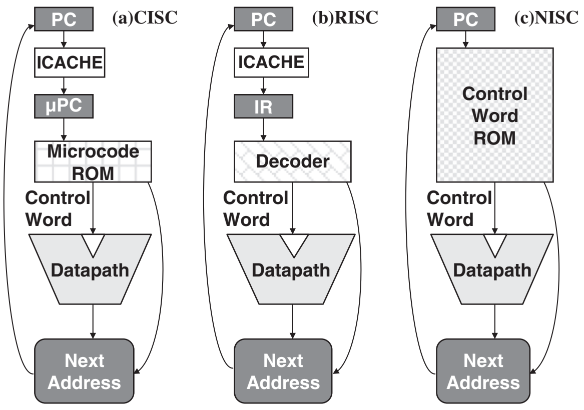


Fig. 4.11 Processor architectures

4.4 Thread Level Parallelism

Multitask OS improves CPU utilization by *thread level parallelism* (TLP), where a task is decomposed into many threads that can be executed concurrently. Using multi-threading, most I/O latency can be overlapped with other threads, greatly improving the system performance.

4.4.1 Multi-Threading

In general, a work spends much time on waiting peripheral I/O response. Since peripheral I/O communication is much slower than the CPU speed, direct memory access (DMA) is often used to handle peripheral I/O communication. When a CPU wishes to send a message to peripheral I/O, it puts data in memory and calls DMA to transfer the data. After transmission, DMA will generate an interrupt to inform the CPU. During transmission, the CPU is idling.

Threads are a way for a code to split itself into two or more simultaneously (or pseudo-simultaneously) running tasks. Assume that an OS has picked a thread to execute once. Sometimes, when this thread is waiting for I/O, or when it has been run for such a long time that a timer interrupt occurs, OS will save its program status in memory and pick another thread to execute. By multi-threading, I/O latency is overlapped with other threads under execution.

Memory stall is the major barrier in processor performance. Video and image applications are memory-dominant. That is, much time is spent on waiting for memory response. Multitask OS cannot switch thread execution when memory stall occurs, because thread switching requires it to save its status into memory, which causes access conflict in the memory.

Simultaneous multi-threading (SMT) [55], or *hyper-threading* [56] techniques were proposed to solve memory latency problem through hardware-supported TLP.

For example, a two-thread SMT processor has two program status words (register file, ALU flag and program counter) and two instruction fetch buffers. These two instruction streams are alive simultaneously. The issue stage selects one instruction to issue into ALU from the two instruction streams once every time. Only when both instruction streams are stalled by memory waiting, ALU is kept busy. Multi-task OS picks two threads for the two instruction streams at once. When a thread needs to wait for I/O, OS saves its program status word into memory and picks a third thread for this instruction stream. From the view of OS, SMT works as a shared-memory multi-processor.

4.4.2 Multi-Processor

A *multi-processor* core used in an embedded system is now very common. As shown in Fig. 2.2, most embedded system contains at least a general purpose processor and a DSP. In these systems, each processor is a standalone design, run on its own task scheduler, or even on its own OS.

There are two multi-processor architectures, *message-passing* and *shared-memory*, distinguished by their communication method.

In a *message-passing* architecture, each processor is working alone. Each memory and peripheral I/O is controlled by only one processor. Processors are connected by a type of communication channel or network. Communications between processors are explicitly described in software. In a *shared-memory* architecture, at least one memory is accessible by two processors. All components are connected by bus; every processor can be a bus master. Communications between processors can be explicit or implicit.

The choice of architecture depends on performance profiling. A memory can only serve one processor at a time. When two processors access to the shared memory simultaneously, one processor should be stalled by bus arbiter. If only some data are shared by two processors, message-passing architecture will have much better performance than shared memory. On the other hand, if a lot of data is shared, such as a big picture, a single memory used to store this picture can save cost and communication time.

In the shared-memory architecture, an important problem that hardware needs to solve is *cache coherence*. A high performance processor always needs a cache to buffer recently-used memory items to reduce memory latency. On a shared-memory multi-processor, a memory location may have multiple copies in these caches. If one processor modifies the memory content, but the result is still in its cache and not written into memory yet, other processors that read the content in this memory location will get a wrong and not up-to-date value. There are two methods to solve this problem. One method sets the shared-memory address area as non-cacheable: processors only cache its local memory. Another method is invalidation: whenever a processor writes to shared memory, all other processors should invalidate the content in their cache and reload it from shared memory when they need to refer to the value. The invalidation can be implemented by the following strategy: every processor is always monitored on the bus; when a write transition is issued by other processor, the relative cache is set invalid.

Multi-processor can be *heterogeneous* having different types of processors inside, or homogeneous with identical processors. Programming for a heterogeneous multi-processor system is difficult. Even by the toolset such as OMAP as described in Section 2.1.1, many things, such as task partitioning, have to be done manually.

In SoC, more and more embedded systems use *homogeneous* multi-processors. On a homogeneous shared-memory multi-processor, a single OS can schedule tasks like an SMT processor. The purpose of using a heterogeneous multi-processor is to handle different types of operations. ASIP breaks the barrier by extending its instruction-set with specific instructions. Thus, an ASIP can process both DSP application and control-oriented application well. Intel XScale and PLX are processors designed for this purpose. XScale is based on ARM processor and adds many features for multimedia. For example, it has saturation arithmetic instructions, MAC instructions, and 40-bit accumulator for long-tap FIR.

In a multi-processor system, inter-processor communication bandwidth dominates system performance. Many communication models had been introduced for high-end multi-processor supercomputers, such as Hypercube, mesh, or linear array.

4.4.3 Massively Parallel

3-D graphics require massively parallel computing. A 3-D object is modeled by many small triangles (or polygons). When this object rotates, all the triangles will rotate by the same angle. 3-D graphics need to compute thousands of polygon operations in parallel. Some high-performance massively-parallel processors had been introduced for gaming machines.

General Purpose Graphic Processing Unit (GPGPU) [57] is a successful massively-parallel processor. It is widely used in desktop PC to accelerate graphic computation. Like ILLIAC-IV, it is composed of many simple processing units optimized for data processing, and shares a control unit to maintain control flow. All inter-processor communications are handled by a compiler on a banked shared memory. Figure 4.12 shows the nVIDIA GeForce 8800 architecture [58] as an example GPGPU.

GeForce 8800 is composed of eight Texture Processor Clusters (TPC). Each TPC contains a texture memory which is a global memory shared by the two Streaming Multiprocessors (SM). Each SM contains eight Streaming Processing Units (SPs) and two Special Function Units (SFUs). A control unit contains a cache and a dispatcher to issue operations to these units. An SM contains a large amount of 8192 registers. Compiler dispatches these registers to threads such that context switch effort is minimized. GeForce 8800 can execute 12288 threads simultaneously.

As shown in Fig. 4.13, the *MIT RAW processor* is an early implementation of Network-on-Chip (NoC) [59] and the Intel 80-core *tera-flops research chip* is another recent example [60]. RAW processor is built of replicating tiles; each contains a MIPS R4000-like general purpose processor and a programmable router.

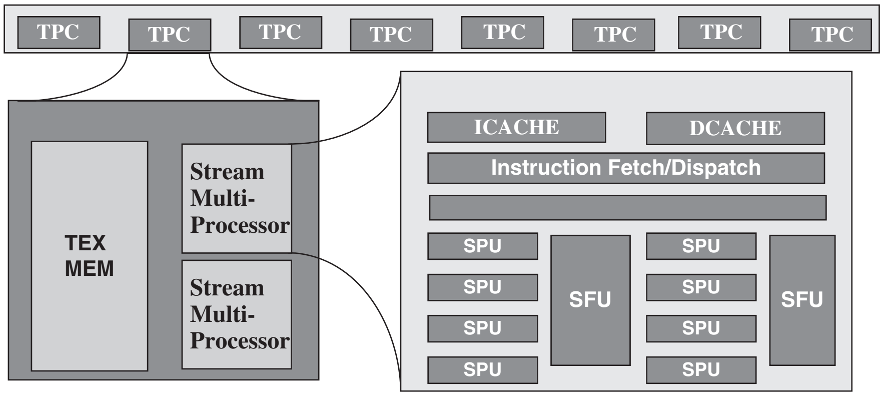


Fig. 4.12 nVIDIA GeForce 8800

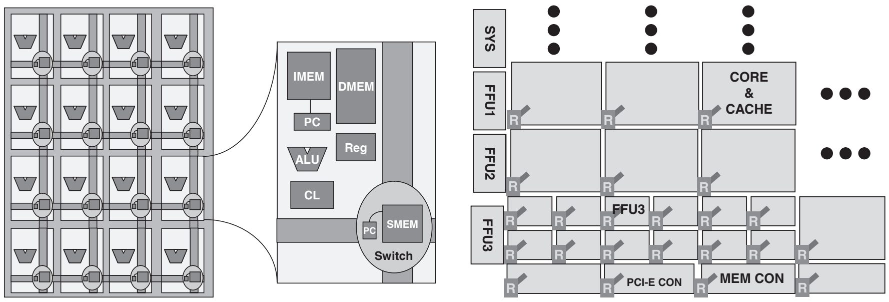


Fig. 4.13 Many cores: (**a**) MIT RAW Processor and (**b**) Intel Tera-Flops chip

Processor networks are used in supercomputers. In off-chip networks, the number of links is determined by the pin-out limitation. Many network topologies such as hyper-cube or star-network have been introduced considering the interconnection cost and broadcast efficiency trade-off. In an on-chip network, this limitation is absent. The topology choice is determined by the wiring physics, because in the nano-meter era, wire delay will exceed gate delay. A long link will take many cycles to transfer. In the shared-bus architecture as shown in Fig. 2.2, the clock skews between a shared memory and the processing units are unable to balance. A *2-D mesh* architecture with shorter interconnection that only propagates data to its neighbor in one cycle becomes necessary for a high-performance many-processor system. On a mesh-based NoC, synchronization cost between two far-away processors is high. Streaming or systolic array programming is more suitable for such mesh-based NoCs [61]. On a mesh-based many-processor system, networking strategy becomes the key factor in system performance rather than processor core. NoC is becoming an important research topic for future tera-scale chips.

Chapter 5
Parallel Compiler

Parallel processing had been developed in 1960s on some high-speed vector processors such as ILLIAC-IV and Cray-1 to increase the scientific computation speed. Since scientific codes contain many one-dimension vector and two-dimension matrix operations, using a vector processor can perform these operations simultaneously on its processing elements. Since then, many parallel compilation techniques have been developed. Some of the parallelization techniques related to the design of a compiler for our PLX processor will be presented in this chapter.

5.1 Vectorization

A *vector* is represented as A[`begin: end: stride`]. The array index is extended to 3 literals to represent the vector operation performing on element `begin`, `begin+stride`, `begin+2*stride`, ..., `end`. When `stride` is 1, it can be omitted. For example, the following code:

$$\text{for(I=1; I<=64; I++)}C[\text{I}]=A[\text{I}+1]+B[2^*\text{I}-1]$$

can be represented as a vector addition

$$C[1{:}64] = A[2{:}65] + B[1{:}127{:}2]$$

In addition to scientific computation, people wish to utilize the vector ability in more fields. To optimize a general algorithm into vector needs in-depth analysis. *Vectorization* technique for sequential code has been widely studied in 1970s. Vectorization technique transforms nested loops into vector by dependence analysis, dependence reduction, and performance improvement.

5.1.1 Dependence Analysis

If there is no semantic difference between executing a loop in a sequential order and executing it as a vector operation, this loop is able to parallelize. A counter example is shown in the following code

DOI 10.1007/978-1-4020-9623-5_5, © Springer Science+Business Media B.V. 2009

$$\text{for(I=1; I<=64; I++)}A[\text{I+1}] = A[\text{I}]+B[\text{I}];$$

On executing the code as a sequential loop, we have the following result: $A[3]_{\text{new}} = A[2]_{\text{new}} + B[2]_{\text{old}} = A[1]_{\text{old}} + B[1]_{\text{old}} + B[2]_{\text{old}}$. On executing it as a vector operation: $A[2{:}65{:}1] = A[1{:}64{:}1] + B[1{:}64{:}1]$, the result will become $A[3]_{\text{new}} = A[2]_{\text{old}} + B[2]_{\text{old}}$, which is different to the result obtained by executing it as a sequential loop; thus the loop is unable to parallelize.

In above example, the operand of the second iteration uses the result of the first iteration $A[2]_{\text{new}}$. In other words, the execution of the second iteration is dependent on the first iteration. Two statements can be executed in parallel only when there is no dependence between them. The statements of the first iteration and the second iteration are dependent, so they cannot be executed in parallel as a vector.

Dependence can be classified into the following four types [62]:

(1) *Flow dependence*, or *Read after Write* (RAW) dependence. If one operand of the second statement is the result of the first statement, the second operation should wait until the first statement finishes.
(2) *Anti dependence*, or *Write after Read* (WAR) dependence. If the second statement overwrites one operand of the first statement, the second statement cannot execute earlier than the first statement to avoid change of operand value.
(3) *Output dependence*, or *Write after Write* (WAW) dependence. If two statements write to the same destination, they cannot execute simultaneously to avoid having an ambiguous result.
(4) *Input dependence*, or *Read after Read* (RAR) dependence. When two statements use the same operand, they are said having input dependence.

Input dependence is not an actual dependence because the statement execution is not dependent on each other. Input dependence is used to group the statements closer such that we can reuse the same operand from the register to save memory load time.

The anti and output dependences can be removed by variable rename technique, thus they are also called *false dependences*, and only the flow dependence is called a *true dependence*.

Therefore, the *loop-carried flow dependence* actually limits vectorization. To precisely determine the loop-carried flow dependence, we can analyze the array index relationship of the statements in a loop [63]. Consider the generalized expression:

$$\text{for(I=1; I<=N; I++)}A[a+b^{*}\text{I}]=f(A[c+d^{*}\text{I}])+\text{g(I)};$$

Where g(I) do not use array A. Relative to the above example, $a = 1$, $b = 1$, $c = 0$, $d = 1$, g(I)=B[I] and $f(\text{A}[x]) = \text{A}[x]$. To analyze a loop-carried flow dependence is to check whether the result $A[a + b^{*}x]$ is used as an operand $A[c + d^{*}y]$ at later iteration. The loop-carried flow dependence exists if and only if there exist integers x and y, $1 \leq x < y \leq N$, such that $a + b^{*}x = c + d^{*}y$.

By number theorem, the equation $a + b^*x = c + d^*y$ has integer solution x, y if and only if $a - c$ is multiple of GCD(b, d), or GCD(b, d)|($a - c$), where GCD is the Greatest Common Divisor. From the above example, GCD(b, d) = GCD(1, 1) = 1, $a - c = 1 - 0 = 1$, GCD(b, d)|($a - c$) is true.

For a loop that contains many statements, we should check whether the statement result is used as operand by any other statement at later iteration or not, that is, we should check the GCD(b, d)|($a - c$) for all statement pairs.

The single loop dependence check can be extended to nested loop. For example, given the following sample code:

```
for(K=1; K<=L; K++)
 for(J=1; J<=M; J++)
  for(I=1; I<=N; I++)A[a0+a1*I+a2*J+a3*K]=f(A[b0+b1*I+b2*J+b3*K]);
```

the dependence checking is performed from the innermost loop to the outmost loop. At a specific outer loop J = x_2 and K = x_3, the innermost loop contains loop-carried dependence if and only if there exist $1 \le x_1 \le N$, $1 \le x_2 \le M$, and $1 \le x_3 < y_3 \le L$, such that $a_0 + a_1^* x_1 + a_2^* x_2 + a_3^* x_3 = b_0 + b_1^* x_1 + b_2^* x_2 + b_3^* y_3$, or $(a_1 - b_1)^* x_1 + (a_2 - b_2)^* x_2 + a_3^* x_3 - b_3^* y_3 = b_0 - a_0$. The integer solution exists when GCD($a_1 - b_1, a_2 - b_2, a_3, b_3$)|($b_0 - a_0$). Similarly, the dependence check equation for the second loop is GCD($a_1 - b_1, a_2, b_2, a_3, b_3$)|($b_0 - a_0$), and GCD($a_1, b_1, a_2, b_2, a_3, b_3$)|($b_0 - a_0$) for the outmost loop.

5.1.2 Loop Normalization

By number theorem, the above GCD test for dependence analysis works only when the loop index begins from 1, ends at a number N, and increases by 1. General loop that does not satisfy this constraint needs to *normalize*.

Given the following code:

```
P=10;
for(J=0; J<100; J=J+2)A[P++]=A[2*J]+J;
```

by performing analysis on the loop argument, we can replace the loop index J to a new index K, that is, $J = 2^*K - 2$, and change the loop index dependent variable P to $P = K + 9$. The loop is transformed into:

```
for(K=1; K<=50; K++)A[K+9]=A[4*K-4]+(2*K-2);
```

Now this loop is normalized and dependence check can be performed.

5.1.3 Loop Transformation

Consider the following code:

```
for(J=1; J<=M; J++)
  for(I=1; I<=N; I++)A[I][J]=A[I−1][J];
```

This code fills the whole array with row 0 in an order of column by column, and the inner loop contains loop-carried dependence. If the code is transformed into:

```
for(I=1; I<=N; I++)
  for(J=1; J<=M; J++)A[I][J]=A[I−1][J];
```

By exchanging the two loops, the new code works row by row. The two results are the same but the later row-wise order can work more efficiently in a vector machine.

Loop transformation procedure sequentially selects a pair of loops which is legal to apply a transformation, and check its dependence by GCD test as described above. If more than one solution is available, the performance or data locality gain is used to help decision making.

Loop transformation is the key technology to improve parallelism and data locality. Many transformations had been introduced [64, 65]. For example, loop skewing helps on systolic array algorithms to utilize memory; loop interchange and reversal helps on linear algebra that contains dense matrices.

5.1.4 Dependence Removal

Instruction Level Parallelism can be improved by removing false dependence. The techniques include variable rename, scalar expansion, node splitting and control flow conversion. The following code is used to explain.

```
     for(I=1; I<=N; I++) {
S1:    v=A[I]+B[I];
S2:    v=v*C[I];
S3:    C[I+1]=v+I
S4:    D[I]=D[I−1]+D[I+1] + 2;
S5:    if(E[I]>F[I])
S6:    R[I]=E[I]−F[I];
S7:    elseR[I]=E[I]+F[I];
     }
```

S1 and S2 are outputs dependent on variable *v* that restricts S1 vectorization. If variables *v* from the result in S2 and the operand in S3 are renamed to *w*, the output dependence is removed. This technique is called *variable rename*. The lifetime of a local variable starts from its value settled, thus renaming it as a new variable will not cause semantic differences.

S1 contains loop-carried output dependence to itself on variable v; this dependence disables S1 to vectorize. Renaming variable v to v[I] removes this dependence. This technique is called *scalar expansion* for it expands a scalar variable into an array. The disadvantage is that it needs to allocate more memory.

S4 contains a loop-carried flow-dependence, where $D[1]$ is modified at the second iteration and then loaded at the second iteration by $D[I-1]$. S4 contains 2 additions, one is vectorizable. The non flow-dependent part $D[I+1]+2$ can be lift to a new statement, and store its result on a new variable $T[I-1]$, then use $T[I-1]$ to replace the non flow-dependent part in the original statement. After that, the new statement becomes vectorizable. This technique is called *node splitting*. The index $I-1$ of the new variable T is aligned to the flow dependence part $D[I-1]$ such that the data shift which is required for ILLIAC-IV array architecture can be performed in parallel.

S6 and S7 have control dependence on S5. The program counter (PC) value set by S5 conditional branch operation is the address of S6 or S7, which will depend on the S5 comparison operation result. It causes the program counter value become ambiguous when all iterations of S5 are executed simultaneously. In other words, S5 has loop-carried output dependence on program counter. To avoid program counter ambiguity, conditional branch operation should be removed. In ILLIAC-IV, each PE contains a *mode* register that can disable the current instruction execution. When a PE is disabled, the relative result keeps no change. The conditional branch execution can be changed to execute all statements with proper vector mask. A new Boolean array is added to store the S5 comparison result. The Boolean array (1-bit for each element) is sent to the *mode* register or *vector mask* register when S6 vector executing, and its complement is sent when S7 vector is executing. The execution with mask expression works as a three-operand one-result operation, the three operands are the original two ALU operands plus the vector mask; such *control flow conversion* [66] technique changes control dependence into data dependence.

The result after dependence removal is shown in the following code:

```
      for(I=1;I<=N;I++) {
S1:     v[I]=A[I]+B[I];
S2:     w=v[I]*C[I];
S3:     C[I+1]=w+I;
S4a:    T[I−1]=D[I+1]+2;
S4b:    D[I]=D[I−1]+T[I−1];
S5:     VM[I]=(E[I] > F[I]);
S6:     R[I]=(VM[I])?(E[I]−F[I]);
S7:     R[I]=(~ VM[I])?(E[I]+F[I]);
      }
```

5.1.5 Strongly Connected Components

Above transformation can be performed more efficiently by applying graph theorem on the dependence graph.

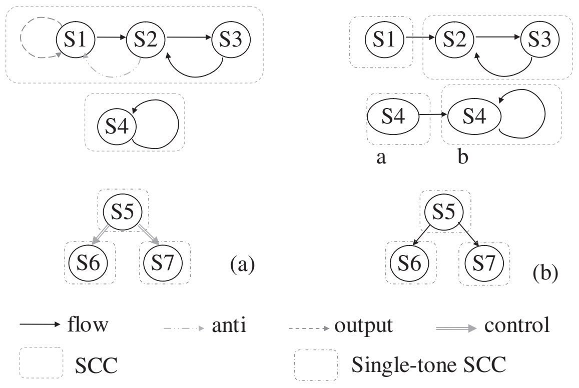


Fig. 5.1 Dependence graphs: (**a**) original and (**b**) after dependence removal

As defined, a dependence graph [67] is a directed graph, whose nodes represent code statements, and arcs are dependences. Figure 5.1 shows the dependence graph of the example in Section 5.1.4.

On a directed graph, a *strongly connected components* (SCC) is defined as: for every pair of nodes u and v if there is a path from u to v and a path from v to u. An SCC can be found by using depth-first search technique [68].

In geometric view, an SCC contains nodes that form a circle. A circle in a dependence graph means that there are loop-carried dependences on these statements, which are not vectorizable. A *single-tone SCC* is defined as an SCC having only one node and no arc to itself; thus, it is vectorizable.

5.1.6 Loop Distribution

If we treat an SCC as a single supernode, the arcs in a dependence graph will have all a forward direction. The loop can be partitioned into many sub-loops on a forward-only path that will not make a semantic difference. All SCCs have to be changed into independent loops, and the original loop headers have to be distributed to these new loops. A single-tone SCC can be directly transformed into a vector. The result of the above example then becomes:

```
S1:  v[1:N]=A[1:N]+B[1:N];
     for(I=1; I<=N; I++) {
S2:     w=v[I]*C[I];
S3:     C[I+1]=w+I;
     }
S4a: T[0:N−1:1]=D[2:N+1:1]+2;
S4b: for(I=1; I<=N; I++)D[I]=D[I−1]+T[I−1];
S5:  VM[1:N]=(E[1:N] > F[1:N]);
S6:  R[1:N]=(VM[1:N])?(E[1:N]−F[1:N]);
S7:  R[1:N]=(∼VM[1:N])?(E[1:N]+F[1:N]);
```

5.2 Simdization

A subword-parallel SIMD processor has more restrictions than a vector machine. For example, a subword-parallel SIMD core is restricted on memory access. Given the following code:

$$\text{for(I=1; I<=64; I++)}C[\text{I}][\text{I}]=A[\text{I}][\text{I}]+1;$$

The memory items are discontinuous. To process above example, PLX has to load the discontinuous memory items $A[0][0]$ and $A[1][1]$ by different load instructions, and pack them into one register to process addition instruction together. While memory access latency is very long, the addition of $A[0][0]$ can finish when waiting $A[1][1]$ to be loaded in a sequential execution scalar processor. Packing the operations would not improve performance, but increase the pack/unpack overhead.

When vector items are continuous, subwords can be loaded together by one load instruction, and memory access count can be reduced. It induces extra effort to handle neighboring data in a subword-parallel SIMD mode.

5.2.1 Control Flow Conversion

Control flow conversion that converts if-else control into execution with mask is introduced in Section 5.1.4. Implement an execution with mask needs three read ports on a register file and three operand ports on an ALU for the extra mask operand, and the register file write port needs to be byte writable; thus, the hardware cost is increased. While mostly control flow will not become the performance bottleneck, increasing hardware cost is not worthy.

A multiplexer behavior, such as $R=X?A:B$, can be implemented using an AND-OR logic as $R=(X\&A)|(\sim X\&B)$. The S6 and S7 statements in the example of Section 5.1.4 can be changed to:

$$\text{S6: } R[\text{I}]=(VM[\text{I}]\&(E[\text{I}]-F[\text{I}]))|(\sim VM[\text{I}]\&R[\text{I}]);$$
$$\text{S7: } R[\text{I}]=(\sim VM[\text{I}]\&(E[\text{I}]+F[\text{I}]))|(VM[\text{I}]\&R[\text{I}]);$$

The two statements can be further optimized as:

$$R[\text{I}]=(VM[\text{I}]\&(E[\text{I}]-F[\text{I}]))|(\sim VM[\text{I}]\&(E[\text{I}]+F[\text{I}]));$$

The new statement only contains basic logic operations that can execute on a 2-operand ALU. But VM[I] is one-bit length and $E[\text{I}] - F[\text{I}]$ are subwords. Before its execution, VM[I] has to expand into a subword size for the bitwise AND/OR operation. This expansion is performed by the subword-parallel comparison operation in S5. Subword-parallel ALU sets every bit in the related subword to 1 (as an integer value -1) when the comparison result is true, and sets to 0 when the comparison is false.

5.2.2 Memory Alignment

For cost and power consideration, most RISC processor requires all memory accesses to be aligned, that is, the data loaded from memory cannot cross the 64-bit boundary if the processor is of 64-bit length. An across-boundary access should be split by a compiler.

Memory alignment becomes more critical when using an SWP-SIMD processor on multimedia applications, where we are asked to pack the memory components into one superword to access together. Although each component does not cross the boundary, the packed one may cross. For example on processing an RGB24 format picture, the packed element is 24-bit (8-bit for each of the Red, Green and Blue components), the third element will cross the 64-bit boundary. Another example is Motion Estimation. This algorithm shifts a search window one pixel at each iteration, making most reference frame accesses misaligned.

Current technology handles misaligned vectors as a stream [69]. Registers are used for each vector as a stream buffer. Vector elements are collected in the registers and shifted to proper aligned position. Figure 5.2 shows the concept. Assume that the data precision is 16-bit, a vector begins from w1, and the vector length is 4. Loading 64-bit from w1 will cross the 64-bit boundary. To avoid the misalignment problem, `Vload` instruction loads two words from w0 and w4, and use `Vpermute` instruction to combine the loaded words. The second word is kept in a stream register for the following vector.

Operands of a vector equation may have different stream shifts. As the following example shows:

$$C[2{:}66] = A[1{:}65] + B[3{:}67];$$

Using the above policy, both streams A and B have to left shift 1 and 3 positions respectively, and the addition result has to right shift two positions. If they have to be aligned to stream C, stream A has to right shift one position and stream B left shift one position, saving one shift operation.

By above discussion, many policies are possible to handle the stream shift.

(1) Zero Shift: It is the same as Fig. 5.2. This policy shifts each misaligned load stream with an offset of zero, and shifts the store stream from offset zero to the alignment of the store address.

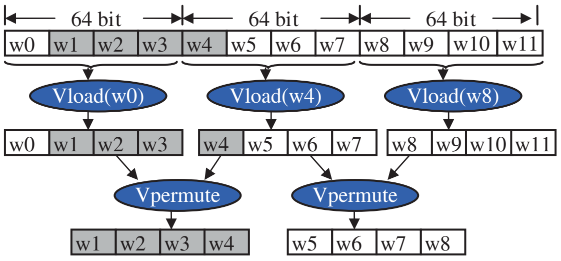


Fig. 5.2 Load vector by streaming

(2) Eager Shift: This policy shifts each load stream directly to the alignment of the store stream.
(3) Lazy Shift: This policy pushes the shift towards the root of the expression tree as close as possible. And
(4) Dominant Shift: This policy shifts to the most frequent alignment position in equation.

5.2.3 Permutation Optimization

Data length conversion can also be handled in streams [70]. When the variables in a statement have different precisions, the load streams have to be unpacked with the largest precision, and the result has to be packed with the same precision as the store variable.

While the subwords of an SIMD instruction are packed into a register, each subword cannot be easily moved alone. In order to unpack four 16-bit subwords in a register with a 32-bit precision, the first subword should right shift 16 bits and the second subword right shift 32 bits to combine into a new register; the third subword is left shifted 16-bit and combined with the forth subword. Totally, 3 shift and 2 combine operations are needed, without including the sign extension.

Many multimedia algorithms themselves contain *permutation*, such as butterfly-order on FFT, or average/difference of two audio channels on MP3. Efficiently handling permutation is not easy. Figure 5.3 shows two implementations of a simple example in MP3 encoder, which calculates the average and difference of the left and right channel samples.

The left channel is the even parts of the audio sample array, and the right channel is its odd parts. The results should also be interleaved into a one-dimension array. Figure 5.3(a) first left shifts samples to a stream aligned on the right channel, then calculates its average and difference, and packs them into the result register. Figure 5.3(b) loads double samples into two registers, packs their even and odd parts, calculates their average and difference, and packs the even and odd results into result register. The first method uses 5 registers and 5 operations to get 4 result samples; the second implementation uses 7 registers and 8 operations to get 8 result samples. The throughput of the second implementation is higher, but it needs more registers, a tradeoff in optimization. Optimizing a code with the fewest permutation instructions can be formulated as a multi-cut problem which is NP-hard [71].

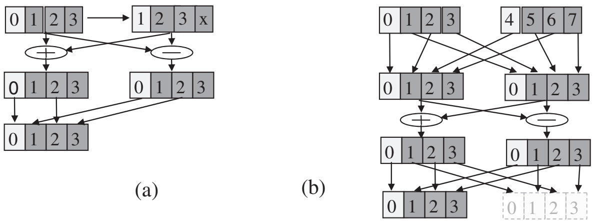


Fig. 5.3 Interleaved average/difference implementations

5.2.4 Subword Fusion

General software contains many non-vector operations. Packing them into a *superword* to process together may increase performance. When addition and subtraction operations are adjacent, negating the subtraction operand and adding them in subword-parallel can improve performance.

MIT University first introduced the concept of *fusioning* these operations [72]. They used a heuristic-based two-cluster partitioning algorithm. Instructions are partitioned into scalar and vector parts. One instruction is moved from the scalar part to the vector part once, and the vector part is re-packed to find the minimum cost. The cost contains pack/unpack overhead.

Vienna University extends the fusion on addition/subtraction pair to increase SIMD utilization [73]. They use depth-first search based sorting method with chronological backtracking to discover SIMD style parallelism in a scalar code block, aiming to reduce the overall instruction count. The addition/subtraction pair finding is considered as reducing number of source operands.

5.2.5 Matrix Transpose

Most processors store array elements in row-wise. Column vector items are not continuous in memory. They should be loaded independently and packed together. Packing four column elements into a superword needs four non-sequential memory `load` and three `pack` operations, which is a large overhead relative to the small code size. To speedup, we can load a 4×4 array from memory into 4 registers, which only needs 4 memory loads. Then transpose the array to get 4 column vectors. Transposition can be efficiently obtained by eight PLX permutation instructions as shown in Fig. 5.4. The first stage exchanges the odd subwords in an even row and the even subwords in an odd row by using two permutation instructions. The second stage exchanges double words of row0/row2 and row1/row3, each takes two permutation instructions.

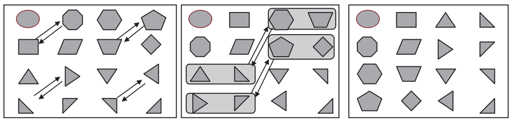


Fig. 5.4 Matrix transpose in SWP-SIMD

5.2.6 Reduction

An extra effort to parallelize multimedia application is to convert summation. Consider the code:

$$\text{for(I=1; I<=N; I++)}s=s+f(A[\text{I}]),$$

which contains loop-carried dependence on variable *s*, vectorization can do nothing. While summation is often used in multimedia, it greatly affects the performance. While SWP-SIMD vector length is short, the loop is able to be partitioned into partial summations such that we can sequentially summarize these partial results at the final stage [74], as shown in the following code:

```
psum[0:VL−1]=0;
for(I=1; I<=N/VL; I++)psum[0:VL−1]+=f(A[I:I+VL−1];
for(I=0; I<VL; I++)s+=psum[I];
```

5.2.7 Loop Unrolling

The vector length of a subword-parallel SIMD processor is short; it can be only 4 or 8 depending on data precision. While a loop iteration count is usually larger than the short vector length, the loop has to be unrolled to fit the short vector length. The example in Section 5.1.4 can be implemented in either one of the following two ways (only the first S1, S3 and S3 statements are shown here):

```
S1: for(I=1; I<=N; I+=VL) v[I:I+VL−1]=A[I:I+VL−1]+B[I:I+VL−1];
        for(I=1; I<=N; I++) {
S2:         w=v[I]*C[I];
S3:         C[I+1]=w+I;
    }
```

Or

```
        for(I=1; I<=N; I+=VL) {
S1:         v[0 : VL−1]=A[I:I+VL−1]+B[I:I+VL−1];
            for(J=0; J<VL; J++){
S2:             w=v[J]*C[I+J];
S3:             C[I+1+J]=w+I+J;
            }
        }
```

The first implementation unrolls the loop after loop distribution, and the second implementation unrolls the loop before loop distribution. The second implementation allocates *v* in a register file, which saves the memory access time for *v*. As semi-conductor technology progresses, the register access time is much faster than the memory one; thus, the performance difference of the two implementations becomes significant.

The second implementation is not always better than the first implementation when it causes data cache swap. If a loop contains many array variables that cannot all fit in a data cache, the partial data of array A that were preloaded in cache (which amount is larger than the vector length) at S1 will be replaced, so it will waste more time to reload array *A* from the main memory at each iteration. This will overcome the gain of register reuse.

The optimal solution for loop distribution is to group all SCCs that are connected by dependence in one loop, but cannot be too large to cause cache swap. The optimization has to compromise between cache strategy and register allocation.

Memory access latency usually affects performance greatly. Software pipelining [75] can be applied to further improve memory access efficiency. With software pipelining, we can reschedule ALU instructions to fill the time when memory load is waiting. By considering memory sequential/ non-sequential accesses and hardware pipeline architecture, and using software pipelining, the performance can improve 34% [76].

5.3 ILP Scheduling

Instruction level parallelism (ILP) scheduling assigns operations into a 2-D slot of spatial and time dimension. ILP scheduling can be divided into cyclic and acyclic scheduling methods. *Cyclic scheduling* works on loop and *acyclic scheduling* works on a basic block code region.

5.3.1 *Software Pipelining*

Figure 5.5 shows one cyclic scheduling method called as *software pipelining*. Assume that this machine is a 3-issue VLIW. A loop of iterations 0 to $n-1$ contains 6 operations from *A* to *F*. Operation *A* is loop-carried dependent to *B*, so A_1 can be executed in parallel with C_0 at the earliest time slot. There are two schedules as shown in Fig. 5.5(a), where the first ALU executes iterations 0, 3, *etc*; the second ALU executes iterations 1, 4, *etc*; and the third ALU executes iterations 2, 5, *etc*. In Fig. 5.5(b), the first ALU executes all *A* and *B* operations; the second ALU executes all *C* and *D* operations; and the third ALU executes all *E* and *F* operations. Schedule (a) has better data locality which is necessary for clustering architecture, but schedule (b) optimizes different functions on the 3-issue ALU.

A_0		
B_0		
C_0	A_1	
D_0	B_1	
E_0	C_1	A_2
F_0	D_1	B_2
A_3	E_1	C_2
B_3	F_1	D_2
C_3	A_4	E_2
D_3	B_4	F_2
E_3	C_4	A_5

(a)

$i = 0$ to n-3

A_0		
B_0		
A_1	C_0	
B_1	D_0	
A_{i+2}	C_{i+1}	E_i
B_{i+2}	D_{i+1}	F_i
	$C_{n\text{-}1}$	$E_{n\text{-}2}$
	$D_{n\text{-}1}$	$F_{n\text{-}2}$
		$E_{n\text{-}1}$
		$F_{n\text{-}1}$

(b)

Fig. 5.5 Software pipelining

In general, data dependences exist in various types. A data may be referenced k iterations later where k is not 1. Then the software pipelining cannot be as simple as above example. Sometimes it requires using a heuristic method, such as *modulo scheduling*, to schedule.

5.3.2 Basic Block Extension

Acyclic scheduling works on a basic block code region. A basic block, as described in Section 2.2.2 on CDFG discussion, has a single entrance at its head and an exit at its tail in a control flow graph. No backward arc is inside a basic block. The code formation is a heuristic process, it selects instructions with data dependence constraint and resource usage conflict, to target optimization of code size or power consumption.

A larger code region has more instructions to select and get better efficiency. The key technique of acyclic scheduling is to enlarge code region. A basic technique is *loop unrolling*. It removes the backward arc in control flow graph, thus the basic block is extended to contain n times of operations.

Tail duplication is another technique to extend basic block. As shown in Fig. 5.6, the control flow graph is partitioned into 4 basic blocks by an `if-else` decision. Duplicating BB4 and moving them into the `if-else` region will reduce the basic block number to 3, but BB2 and BB3 are enlarged.

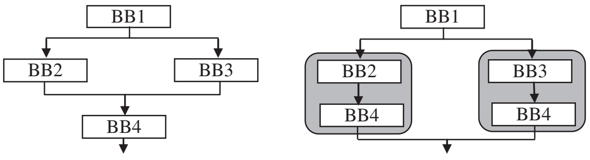


Fig. 5.6 Basic block tail duplication

5.3.3 Speculation

Control flow always limits the instruction stream to fill ILP wide spatial slot. Sometimes if we know by profiling that a branch has a higher probability to execute, it must be executed in parallel with a current basic block. The speculation technique will bring this branch ahead before the check point to improve parallelism, and the execution will be recovered if speculation result is negative.

Preload is a frequently used speculation method. A memory load is able to execute whenever the `load/store` unit is not in use. If the control flow branches to another path, the loaded data is just waste but will not affect the result. In moving more instructions before branch, more registers are required to store these temporary results, which reduce the number of available registers in the original basic block.

5.4 Threading

Multitask OS scheduling is maintained on two levels: process and thread. A *process* is a standalone program. Killing a process during scheduling will not affect other processes. Process has its own heap and stacks memory, file handler, and so on. Synchronization between processes is seldom. *Thread* is a piece of process execution stream. Threads are not independent, killing a single thread may cause process execution failed. Each thread has its own stack, but the heap memory and file handler are shared with others.

Threaded programming offers software portability for parallelization. On a serial machine, threads can work concurrently by time sharing; on a massively parallel machine, threads can work in parallel simultaneously. The difference of scheduling is taken care by OS. To achieve portability, a standard to handle threads is necessary.

5.4.1 Profiling and Analysis

In a general code, 90% of the execution time is spent on 10% of the code. *Profiling* is used to tell the programmer where the performance bottleneck is. The result of profiling is some statistical information on a code, such as execution time, subroutine call statistics, operations used, and memory access time.

Static profiling works by analyzing the representation of a program code without executing it. The non runtime environment gives the possibility to go into greater detail in the analysis but also places restrictions on it. Non deterministic properties, such as recursion, dynamic data structures and non bound loops, in a code region cannot be estimated without running data from the input, which in turn requires dynamic profiling.

Dynamic profiling on the other hand executes a code with a given testbench instead of analyzing it. During execution the profiler gathers a code which is being executed and whatever properties of the execution are deemed interesting. The dynamic profiling cannot give the engineer as profound information on the code as the static profiling does, but it can in detail report what happens during execution of the code with a well-defined set of inputs.

In order to discover parallelism, data dependence is one of the most important characteristics in a code. A code can often be clustered by its spurious dependences; for example, the accesses of two memory objects may be conflicting, if the objects cannot be proved independent. A single spurious dependence can prevent multiple opportunities for parallel execution. Analysis clarifies the code picture either by finding precise data dependences or by removing spurious ones to improve parallelism [77].

The chief obstacle to discovering opportunities to parallelize a multimedia application is identifying dependences between pointer references. A high-quality pointer analysis is essential in determining the relationship between pointer references. However, there are many coding constructs and programming practices that veil the true picture of memory usage from pointer analysis. For some of these cases, like recursive data structures and arrays, more specialized analyses such as shape analysis and array analysis will be very helpful in clarifying the picture.

Pointer analysis determines what objects a memory reference can possibly access. Heap-sensitive pointer analysis finds whether the allocation function for a particular type of dynamically-allocated memory object is frequently reused to allocate multiple objects. Such kind of code reuse is a must to distinguish objects that share a static allocation site. Field-sensitive pointer analysis will group together all of the objects pointed to by a structure. This prevents the compiler from distinguishing objects through those pointers. This case appears regularly since multimedia programs commonly manipulate multiple data channels, and programmers use structures to organize data hierarchically.

Array analysis can indicate whether or not the pointers really refer to the same memory location, when two pointers are known to refer to the same object. This form of analysis conveys information about which loop iteration carries a data dependency. Array analysis can also determine whether different loops access the disjoint subsets of a given object. Finally, array analysis can be used to derive the data correlation between iterations of separate loops.

One important aspect of multimedia applications is that they often have a range of supported sample rates, sizes, or resolutions and use many symbolic variables in the interest of code reuse. Dimensions determined at runtime create non-affine expressions and variable loop bounds, which stymie many simple array disambiguation tests. In these cases, value constraints analysis can be obtained or computed to assist the array disambiguation.

Value constraint analysis finds the information about the possible range or other constraints on a value, it can be critical in evaluating symbolic tests. Many variables in a code have a relatively small set of values during the majority of code execution, restricted by control flow tests or written constants.

Value relationship inference helps to know the relationship between the values of different variables. Often, one variable is used to compute the value of several other variables. When related variables appear in an index expression, symbolic analyses typically lose precision unless they know the relationship between the variables. These relationships are found by tracking values back through def-use relationships to find common terms. This requires inter-procedural expression computation through memory objects, often dynamically-allocated, to find the relationships between values [78].

5.4.2 Pthread

IEEE standard 1003.1c specifies POSIX (Portable Operating System Interface for Unix) as threaded API in 1995. A thread implementation that follows this standard is referred to as *Pthread*.

The POSIX Pthread function API can be classified into three classes. The first class works directly on threads, including thread creating, detaching, joining, *etc*. These are adjoined by thread attribute functions that set or modify attributes of the threads, (joinable, scheduling *etc*). The second class works on mutual exclusive (mutex) context switch creating, destroying, locking and unlocking. These are supplemented by mutex attribute functions that set or modify attributes associated with mutexes. The third class works on condition variables creating, destroying, waiting and signaling. These are accompanied by condition variable attribute functions that modify or set attributes in condition variables.

The `pthread_create` API registers a thread function in the OS scheduling list. On success, a unique identifier is associated with this thread. And attributes used to describe stack size and schedule policy are created.

The `pthread_join` API waits for the thread specified by the unique identifier to finish.

In shared-memory inter-thread communication, ensuring the shared data correctness is the most important effort on thread synchronization. Pthread defines some APIs for mutex.

When a communication buffer is created in a shared memory, a `pthread_mutex_init` API should be called to initialize a handler for the buffer. The producer should lock the handler by using a `pthread_mutex_lock` before the data is put into the buffer. If this handler is already locked by other thread, the `pthread_mutex_lock` will enter a sleep state until the handler is unlocked. The consumer should lock the handler before checking the buffer status. If a lock is successful but the buffer is empty, it means that the producer is not ready to put data. The consumer should unlock the handler and go sleep. The producer should call a `pthread_mutex_unlock` to wakeup the consumer.

A special locking strategy is *recursive mutex*. A recursive mutex allows a single thread to lock a mutex multiple times. Each lock increases the handler lock counter, and each unlock decreases it. Any other thread can lock the handler only when the lock counter is zero. A recursive mutex is often used in a recursive function, such as a binary search.

A *condition variable* is a data used for synchronizing threads. This variable allows a thread to block itself until a specified data reaches a predefined state. The `pthread_cond_wait` API waits another thread to set the condition variable by the `pthread_cond_signal`. A thread is idle when waiting for a condition variable; the hardware should generate interrupt to wake this thread up when the `pthread_cond_signal` is executed.

A conditional variable is used to support the multi-read lock strategy. In some applications, a data is read by many consumers; for example in matrix multiplication, an element is propagated to all threads that will be operating on this element with a row or column to generate a result. Using mutex, all threads should get this element in sequence. When a thread wishes to write data to a buffer, it should make sure that no reader is using this buffer. When a consumer locks the buffer, the lock counter is increased. When the buffer is unlocked, the consumer broadcasts a condition variable. The producer is woken up by this condition variable, and checks whether the lock counter is zero.

Mutex is a special message-passing type of communication method that passes messages through a shared-memory. There are various methods to implement message-passing in a communication channel, it can be a first-in first-out (FIFO) buffer, a serial bus, or a network. The *message passing interface* (MPI) standardizes these APIs, by hiding the communication protocol under library implementation.

The basic MPI routines are `MPI_Send, MPI_Recv` and `MPI_Bcast`, which are respectively used to send a message to a destination, receive a message from a source, and broadcast a message to others in the group.

5.4.3 Structuring

S*tructuring* the threads of a task helps to maximize concurrency and minimize synchronization effort. Some structure patterns [79] used to parallelize a code are presented in the following.

The basic structure is *parallel threading*. Typically parallel threads are decomposed from independent loop. When each iteration of a loop depends on different data, they can be separated into threads and executed in parallel via loop distribution as shown in Fig. 5.7(a).

When one loop produces a data that will be consumed by a following loop, and each iteration of the following loop only depends upon a limited and known number of iterations of the previous loop and does not overwrite the first loop's input data, it is possible to execute part of the two loops in parallel as long as the real data dependences are respected. Figure 5.7(b) shows such an example.

The second structure is *divide and conquer*. Recursive algorithms such as binary search is an example of divide and conquer, which main thread creates two child threads to search the two parts of an input data base. Each child thread also creates two child threads until the search range is small enough.

The third structure is *decomposition from data*; Figure 5.8 shows an example. This algorithm has five tasks, ABCDE. Tasks ABC process the even and odd parts of an input data. Task D computes the high and low sub-bands of each C's results. Task E summarizes task D's results. This machine has four processors; thus keeping four threads alive is the most efficient way. The input data is decomposed into four parts, even-higher, even-lower, odd-higher and odd-lower. The main thread creates four child threads to process the four parts simultaneously. Main thread is suspended by `pthread_join`, so only four threads are alive. The `pthread_join` after task C guarantees that all data used in task D are all ready.

The fourth structure is *pipelining*. This kind of structure can be derived by using the*software pipelining* technique as depicted in Fig. 5.5. Load balance is

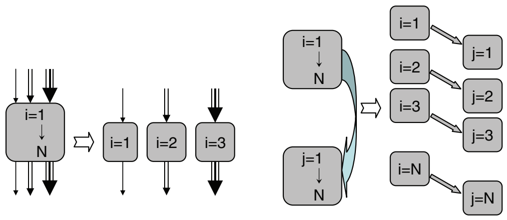


Fig. 5.7 Loop parallelism

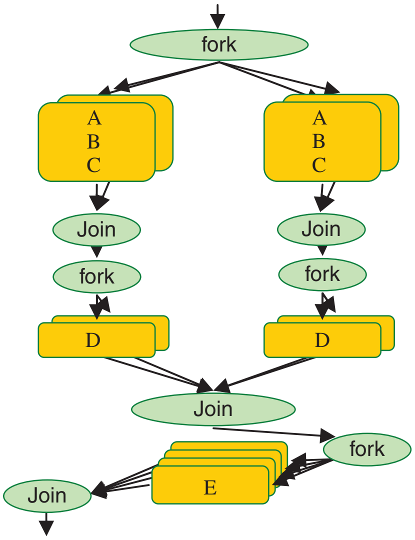


Fig. 5.8 Data decomposition threading

a challenge in pipelining structure; it is restricted by loop-carried dependence. Figure 5.9 demonstrates an example. Figure 5.9(a) shows the data dependency graph of a loop body, where a loop-carried dependence is represented by a backward arc. Due to the existence of this arc, the graph has to be partitioned into three partial functions: A, B and C. Figure 5.9(b) shows that the iterations of all functions are partitioned into threads. Since function B should be processed in sequence, the threading needs more synchronization. Thread B0 should wait for threads A1 and B0 to finish by a `pthread_join` before it can run.

Structuring the sequential function B into parallel threads only increases thread handling effort without improving performance. The pipelining structure is shown in Fig. 5.9(c). The loop body is partitioned into three sub-loops for functions A, B and C by loop distribution, and assigned into three threads. Condition variable is used to synchronize. Iteration B1 waits for A1 to send the shared data using an MPI or a condition variable signal. Iteration B1 does not need to wait for B0 because they are executed sequentially. Now, the local variables of each function are not necessary to be expanded into parallel threads, so the threading cost is lower.

Most threading structures are used to decompose a large input data such that all threads can perform the same computation on different data areas. The data is often a multi-dimension array that can be decomposed into multi-dimension grids. Programmers typically prefer to use a *Single Program Multiple Data* (SPMD) coding style that allow using threads to run the same code in order to save instruction cache loading cost. Each thread needs a mechanism to distinguish its data grid. On

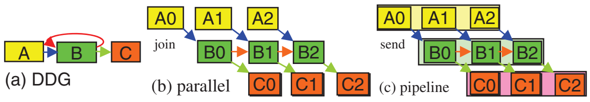


Fig. 5.9 Pipeline thread structure

```
int a[3][4], sum;
main()
{ pthread_t threaded[12];
   sum=0;
    for(int i=0;i<12;i++){
      pthread_create(&threaded[i],NULL,child_thread,
      (void *)i);
    }
   for(int i=0;i<12;i++)pthread_join(threaded[i],NULL);
}
void * child_thread(void *param)
{
    int x=(int)param % 4;
    int y=(int)param / 4;
    sum+=a[y][x];
}
```

Fig. 5.10 SPMD code example

thread creating, a parameter is put on its stack header as an argument; thread can use this parameter to identify its grid location. Figure 5.10 shows a simple SPMD example. The main thread passes a loop-index argument to the `child_thread`, and the `child_thread` will use this argument to identify the location of a 3×4 array.

5.4.4 OpenMP

The structure of a threaded code is different to a sequential code. An algorithm is usually described in a sequential format, such as a linear or binary search algorithm. Programmer should manually reconstruct it into threads. Some compilers support directives to automatically maintain threads. These directives are added within a sequential code to indicate compiler to handle thread and shared data. A compiler not recognizing these directives can compile this code in its original sequential format,

```
int a[3][4], x, y,sum;
main()
{#progma omp parallel num_threads(12) private(x, y)\
               shared (a) reduction(+: sum)
    { sum=0;
      for(int i=0;i<12;i++) {
                x=i % 4;
                y=i / 4;
                sum+=a[y][x];
            }
    }
}
```

Fig. 5.11 OpenMP code of SPMD

so programmer only needs to maintain a code for both sequential and parallel formats. The recent directive standard is OpenMP.

The OpenMP directive starts by adding a keyword "`#progma omp`" on a `structure` block. The code in Fig. 5.11 shows an OpenMP version of the SPMD code listed in Fig. 5.10. It was declared to structure this loop body into twelve threads, where variables x and y are local variables to each thread and array a is a shared variable. Modifying a shared variable will generate mutex automatically. The `reduction` function indicates that the shared variable sum gathers a summation. Instead of a sequential mutex, OpenMP groups threads to generate a partial summation to improve performance.

5.5 Compiler Technique

In this section, we delineate the various phases of a standard compiler and three kinds of popular intermediate representations used for optimization.

The main phases of a standard compiler can be divided into *front-end*, *intermediate code generator*, and *back-end* phases as depicted in Fig. 5.12 [80]. A compiler is a program that can read a source code written in a language, typically a high-level language like C or Fortran, and translate it into an equivalent code in another language, an assembly language usually. The compilation process can be divided into three phases: front-end, middle-end, and back-end. The *front-end*, usually called the *analysis* part, breaks up the source code into constituent pieces and imposes a grammatical structure on them. The front-end also collects information about the source code and stores it in a data structure called symbol-table. Figure 5.12 can be further divided into seven sub-phases. The font-end consists of four sub-phases: *lexical analyzer, syntax analyzer, semantic analyzer*, and *intermediate code generator*. The *back-end*, usually called the *synthesis* part, constructs a desired target code from the intermediate representation (IR) and the information in the symbol-table. The back-end consists of three sub-phases: *machine-independent code optimizer*, *code generator*, and *machine-dependent code optimizer*.

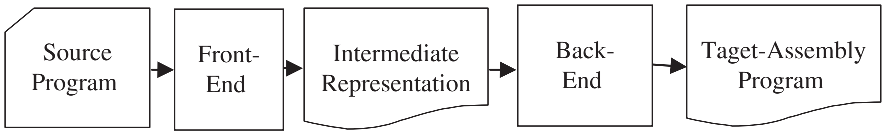
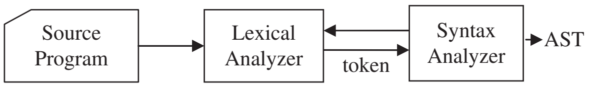


Fig. 5.12 Main phases of a compiler

5.5.1 Lexical Analysis

Lexical analysis, also called scanning or lexing, is the first phase of a compiler. It interacts with the source code and the syntax analyzer as shown in Fig. 5.13. The lexical analyzer reads a stream of characters from the source code statements and

Fig. 5.13 Communication between lexical and syntax analyzers



groups the characters into tokens, and then passes these tokens to a syntax analyzer. Both lexical analyzer and syntax analyzer work together in a producer-consumer relationship. The syntax analyzer calls a lexical analyzer to get tokens and the lexical analyzer provides tokens. *A token* consists of keyword, operators, identifiers, literal strings, punctuation symbols, and so forth. A lexeme is a sequence of characters matched by a given pattern associate with a token. The rule describes a set of lexemes for a particular token is called pattern. For each lexeme, the lexical analyzer produces a token consisting of two parts: token name and attribute value.

For example, suppose that a source code contains the following assignment statement:

$$\text{pos=init} + \text{rate}^{*}100;$$

where "pos", "=", "init", "+", "rate", "*", "100", and ";" are all lexemes. Their corresponding tokens shown in the form <token name, attribute value> are <ID, pos>, <=, >, <ID, init>, < +, >, <ID, rate>, <*, >, <INT-LIT, 100> and <;, >. As we can see, tokens "=", "+", "*" and ";" have just a token name without an attribute value. After lexical analysis, the assignment statement is transferred to a sequence of tokens: <ID, pos>, <=, >, <ID, init>, < +, >, <ID, rate>, <*, >, <INT-LIT, 100> and <;, >.

5.5.2 Syntax Analysis

Syntax analysis also called parsing is the second phase of a compiler. A syntax analyzer uses the tokens produced by the lexical analyzer to create a tree-like IR called *abstract syntax tree* (AST) that depicts the grammatical structure of a token stream.

The syntax of a programming language can be specified by a *context-free grammar* or a *Backus-Naur Form* (BNF) notation. Figure 5.14 shows a representative, but very limited grammar scheme for translating expressions and statements to construct a syntax tree, which we will discuss later. All the nonterminals in a grammar have an attribute *n*, which represents a node in a syntax tree.

The parsing methods typically used in compilers can be classified into top-down and bottom-up. The *top-down* method builds a parse tree from the top (root) and derives its leaves to the bottom, while the *bottom-up* method starts from the leaves and reduces to the root. Both the top-down and bottom-up parse trees should be identical if the grammar is unambiguous. For example, the parse tree of the statement "pos = init + rate * 100;" is illustrated in Fig. 5.15.

stmt	→	expr ";"	{ return expr.n; }
expr	→	rel "=" expr1	{ stmt.n = new Eval(expr1.n); }
	|	rel	
rel	→	rel < add	{ rel.n = new Rel('<', rel1.n, add.n); }
	|	rel <= add	{ rel.n = new Rel('<=', rel1.n, add.n); }
	|	add	
add	→	add "+" term	{ add.n = newOp('+', add1.n, term.n); }
	|	Term	
term	→	Term "*" factor	{ term.n = new Op('*', term1.n, factor.n); }
	|	factor	
factor	→	id	{ factor.n = id.n; }
	|	num	{ factor.n = num.n; }

Fig. 5.14 Grammar for simple arithmetic expressions

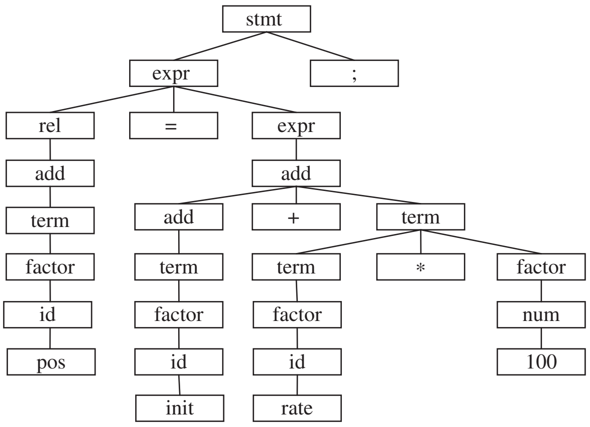


Fig. 5.15 Parse tree for "pos = init + rate * 100;"

A *parse tree* or a concrete syntax tree is a tree IR that represents the hierarchical syntactic structure of a source code. In a parse tree, interior nodes represent non-terminals of the grammar while leaf nodes represent terminals. In Fig. 5.15, `stmt`, `expr`, `rel`, `add`, `term`, `factor`, `id` and `num` are nonterminals that are helpers to represent programming constructs. Leaf nodes are lexemes such as `pos`, `init`, `rate`, "`100`", ";", "+", "*" and "=".

5.5.3 Abstract Syntax

An *abstract syntax tree* (AST) is distinct from a parse tree [80, 81]. As mentioned above, in an AST, the interior nodes represent programming constructs; while in a parse tree, the interior nodes represent nonterminals of a grammar. An AST captures the essential structure of an input in a tree form, while omitting unnecessary syntactic details. An AST is distinguished from a parse tree by dropping the tree nodes in a parse thee that represent punctuation marks, such as the semicolon nodes that terminate statements and the comma nodes that separate function arguments. An

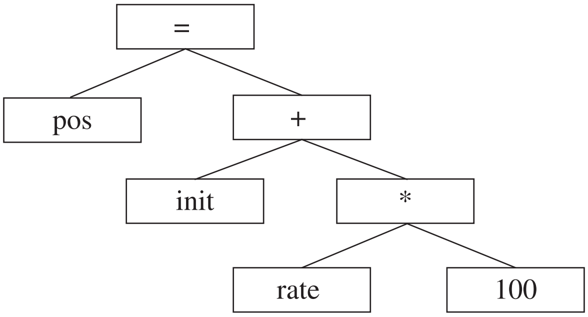


Fig. 5.16 AST for "pos = init + rate * 100;"

AST also omits tree nodes that represent unary productions in the grammar. Such information is directly represented in an AST by the structure of the tree.

Figure 5.16 depicts an AST for "pos = init + rate * 100;". An AST provides a more clear view in comparison to a syntax tree as shown in Fig. 5.15. As mentioned above, an AST omits those unnecessary syntactic details and punctuation marks.

5.5.4 Semantic Analysis

The *semantic analysis* uses AST and the information in the symbol-table to check a source code for semantic error. The main tasks of this phase are listed as follows. First, type checking checks whether the types of two source operands match with each other or with the type of their operator. For example, there is a type mismatch error such as x > "a" while x is defined as an integer variable. Second, it also gathers the type information saved in the symbol-table for the subsequent passes, such as the intermediate-code generation pass. Third, it is likely to perform type conversion or coercion when the two operands of an operator have different types such as integer and floating-point.

Figure 5.17 illustrates an AST for "pos = init + rate * 100;" after semantic analysis. We assume that pos, init and rate are declared as floating point variables. In such situation, semantics analysis should be coercion transformed from integer "100" to floating point such as "100.0".

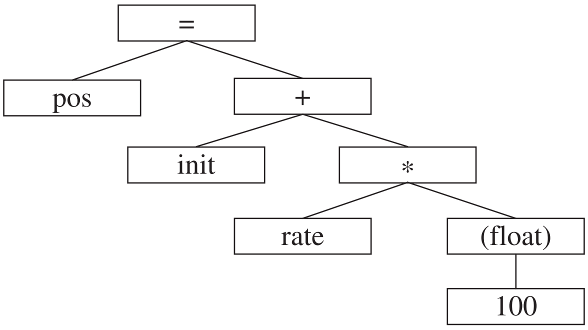


Fig. 5.17 AST after semantic analysis

5.5.5 Symbol-Table Management

Symbol-table is a database in a compiler, which contains information on subroutines and variables [80, 82]. A symbol-table is indexed by a key field, typically a subroutine or variable name, and a record field. The record field is an entry in the database that includes the subroutine or variable name, its type, its position in storage, its scope and so on. Symbol-table usually needs to support multiple declarations of the same identifier within a code.

A symbol-table should support some functions to maintain its database. The maintenance functions in a symbol-table are procedures used to: insert new entry in the table, find a subroutine or variable, delete a subroutine or variable, and so forth.

An appropriate symbol-table manager must have the following four characteristics. First: speed. Look up or insert an identifier into a symbol-table must be as fast as possible. Second: convenience of maintenance. The data type is undoubtedly the most complex data structure in a compiler. It must be well organized for programmers to use other than the compiler writer. Third: flexibility. For example, a good symbol-table infrastructure must not be fixed size with limited identifier number while the input C language contains an arbitrary variable declaration number. Fourth: duplicate entries must be supported. Many programming languages support multiple variable declarations in different scopes with the same name.

5.5.6 Intermediate Representation

The *intermediate representation* (IR) plays an important role in the process of compilation. An IR serves as a bridge between the source code programming language and the target assembly language. Therefore, an IR can be seen as a model assembly language, optimized for a nonexistent, but ideal, computer called a virtual machine. An appropriate IR has the following two characteristics. First: flexibility. A good IR should be easy to produce and to be translated into a target machine code. A *retargetable compiler*such as GCC provides several front-ends for different source code programming languages and several back-ends for different target machines [83]. A retargetable compiler can improve compiler reusability. When adapting a retargetable compiler to a new source language, we just need to develop a new front-end for this source language instead of the whole compiler. Similarly, when adapting it to a new target machine code, we just need to develop a new back-end. Second: ease of optimization. Machine-independent optimization passes exploit the IR to optimize thereby improve the resultant target assembly code.

IRs can be divided into following two categories: Trees that include ASTs and DAGs, and linear representations such as three address code (3AC). An AST depicts the hierarchical structure of a source code. On the other hand, a DAG is a tree structure but more succinct, which gives the compiler more important clue regarding the generation of efficient code by identifying common sub-expressions in an expression or among expressions. With regard to the three address code IR; each instruction has no more than three operands and one operator that can be described as quadruples: two sources, an operator and a result as shown as follows:

$$x = y \; op \; z,$$

where x, y and z are variables while *op* represents any operator *e.g.*, arithmetic operator.

5.5.7 Code Optimization

The term *optimization* in compiler design refers to the attempts that a compiler wants to make a code more efficient than its previous version. In modern times, the optimization of a code performed by a compiler has become more and more important and of course difficult to develop. Implemented on a multi-core machine, a compiler has to face the problem of multiprocessor parallelism.

Compiler optimization must meet the following design objectives [80]. First, the optimization must be correct, that is, must preserve the meaning of the source code. It goes without saying that the first goal is the most important. Second, the optimization must improve the performance of a code. The definition of performance may include execution time and power consumption. Third, the compilation time must be kept reasonable. We need to keep the compilation time short to support a rapid development and debugging cycles. Fourth, the engineering effort required must be manageable. A compiler is a complex system, therefore we must keep the system simple to assure that the engineering and maintenance costs of the compiler are manageable.

Code optimization usually can be performed on the IR. It can be divided into two categories. One is machine-independent optimization; the other is machine-dependent optimization. *Machine-independent optimization* means code transformations that improve the IR without taking into account of any target machine. On the contrary, *machine-dependent optimization* applies transformations according to the target machine.

There are several machine-independent optimization techniques as listed in the following. Control flow analysis technique identifies the flow path of a whole code. Data-flow analysis technique is a technique for gathering information about the use circumstance of variables. Copy propagation technique is the process of replacing the occurrences of targets in direct assignments with their values. Constant folding is the process of simplifying constant expressions at compile time. Common subexpression elimination technique is the process that searches for instances of identical expressions, and analyzes whether it is worthwhile replacing them with a single variable holding the computed value. Dead code elimination technique is the process used to reduce code size by removing code which does not affect the code.

Machine-dependent optimization technique such as instruction scheduling is a process used to improve instruction-level parallelism, which improves the performance of a machine with instruction pipelines. Register allocation is the process of multiplexing a large number of target code variables onto a small number of CPU registers. Vectorization technique is the process of converting a computer code from a scalar implementation, which performs one operation on a pair of operands at a time, to a vectorized code where a single instruction can perform the same operation on multiple pairs of vector operands.

5.5.8 Code Generation

A*code generator* takes an input IR produced by the front-end along with its relevant symbol-table information, and maps it into a semantically equivalent target language.

Code generation includes three primarily tasks: instruction selection, register allocation and assignment, and instruction ordering. *Instruction selection* is the task that transforms an intermediate representation of a source code into a target code. Typically, instruction selection is implemented with a backward dynamic programming algorithm which computes the optimal group for each point starting from the end of the code. In addition, it can be implemented with a greedy algorithm that chooses a local optimum at each step. *Register allocation and assignment* is the task of multiplexing a large number of target code variables onto a small number of CPU registers. Since the number of variables in a typical code is much larger than the number of available registers in a processor, the contents of some variables have to be spilled into memory locations. However the register efficiency is far from memory. On the other hand, the cost of such spilling should be minimized by spilling the least frequently used variables first, but it is not easy to know which variables have been used the least. For this reason, the goal of register allocation and assignment is to keep as many operands as possible in registers to maximize the execution speed of a software code. Register allocation is an NP-complete problem. In this area, there are abundant of research subjects to mention. *Instruction ordering* is the task used to increase instruction-level parallelism by rearranging the order of instructions to avoid pipeline stalls. Though it can improve performance on machines with instruction pipelines, instruction ordering is still an NP complete problem.

5.6 Compiler Infrastructures

The heavy vectorization and SIMDization efforts involve many compilation techniques. Instead of designing their own compiler, people prefer to use an existing compiler infrastructure to help on their work. In this section, we survey four public domain compiler infrastructures: LCC, GCC, SUIF, and IMPACT. Note that in this section (and also in the whole book), we use the word “code” to represent an application program or a segment of program given as an example throughout this book, and the word “program” to represent a program in the compiler infrastrutures.

5.6.1 LCC Compiler Infrastructure

LCC stands for *Local C compiler* or *Little C compiler* [84]. It is a small free retargetable compiler for the ANSI C programming language. LCC uses well established compiler techniques similar to any other compiler. One of the benefits of LCC is its being very compact in size. The total source code size of LCC is around 20000 lines. It is much smaller than many other open-source compilers. The front-end performs lexical, syntactic, and semantic analyses, and some machine-independent optimizations. Both the lexical analyzer and the recursive-descent parser are written by the authors from scratch, which makes it more efficient than a Lex- or a Yacc-based implementation. Its scanner and parser translate a C code into syntax trees. Then, the syntax trees are translated into LCC intermediate representations (IRs) in the form of direct acyclic graphs (DAGs). The DAGs will be dismantled into trees to ease code generation. Then code generation and register allocation are performed on the trees. The code generator matches an IR tree in a code with the fragment patterns of individual instructions and picks the best matches according to the overall tiling cost of the tree. The algorithm can generate an optimal matching for IR trees in linear complexity. Register allocation in LCC uses simple labeling method instead of graph coloring. Considering simplicity, LCC does not perform extensive code optimization in its back-end; the quality of its output code is poorer. The output code quality of LCC, in terms of code size and execution speed, is inferior to that of GCC by an average of 10%. The lack of optimizations in LCC can be partially made up by the post-pass optimizations. Theoretically, this approach complicates future changes for a standardized language like ANSI C, and there have been few lexical or syntactic errors. Indeed, since less than 15% of the LCC program statements is concerned on parsing, the error rate in that code is negligible. Despite its theoretical prominence, parsing is a relatively minor component in LCC as in other compilers; semantic analysis, optimization, and code generation are the major components and account for most of the code and thus most of the errors. The target-independent front-end and a target-dependent back-end are packaged in a single code, tightly coupled by a compact, efficient interface. The interface consists of a few shared data structures, seventeen functions, and a 36-operator DAG language. Retargeting LCC requires rewriting these three back-end components. In practice, new back-ends are implemented by writing new rules and editing copies of an existing configuration and set of interface functions.

5.6.2 GCC Compiler Infrastructure

GNU Compiler Collection (GCC) is a set of programming language compilers produced by the GNU Project [85]. GCC has been ported to more kinds of processors and operating systems than any other compiler. GCC is often the compiler of choice for developing software that is required to execute on a plethora of hardware. Differences in native compilers lead to difficulties in developing code that can be compiled correctly by all these compilers and in building scripts that will run for all the platforms. In GCC, the same parser can be used for all platforms, so if the code can be compiled by a compiler on a platform, chances are high that it can be compiled by others on different platforms. GCC's external interface is generally standard for a UNIX compiler. Users invoke a driver program named `gcc`, which interprets command arguments, decides which language compilers to use for each input file, runs the assembler on their output, and then possibly runs the linker to produce a complete executable binary. Each of the language compiled is a separate program that takes in a source code and produces an assembly code. All have a common internal structure. A per-language front-end parses the source code in that language, and produces an abstract syntax tree and a back-end that converts the trees to GCC's Register Transfer Language (RTL). After compiler optimizations, static code analysis techniques, and a compiler directive which attempts to discover some buffer overflows, are applied to the code. Finally, assembly language is produced.

Figure 5.18 illustrates an overview of GCC which can be divided into three main components: front-end, middle-end and back-end. There are two C-like machine-independent IRs: GENERIC and GIMPLE, and one LISP-like machine-dependent IR: Register Transfer Language (RTL).

GENERIC representation is the main interface between a front-end and the rest of the compiler. Once the source code is parsed and validated, the front-end converts the parsed results into GENERIC, a high-level tree representation where all

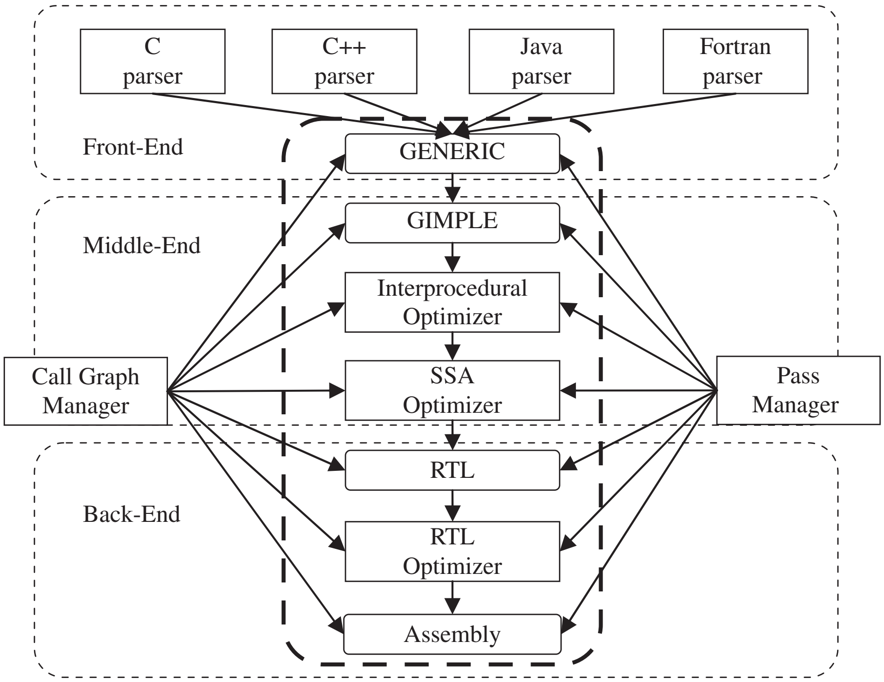


Fig. 5.18 An overview of GCC

the language specific features are explicitly represented. GIMPLE, a three address representation, is a simplified version of GENERIC used to facilitate the job of optimization via restricted grammar.

RTL will contain more detailed machine-dependent information and may vary from one processor to another. RTL is designed to abstract hardware features such as register classes, memory addressing modes and code generation mappings between the back-end and the target processor, which are defined in a *machine description* (MD) file. In the GCC compilation process, the optimizations can be divided into three categories: *interprocedural* optimizer and *Single Static Assignment* (SSA) optimizer operate on GIMPLE IR while RTL optimizer operates on RTL IR. *Introprocedural optimizer* includes inlining and constant propagation,*etc*. The *SSA optimizer* includes vectorization, various loop optimizations, and traditional scalar optimizations such as dead code elimination, *etc*. RTL optimizer includes register allocation, instruction recombination, and so forth, which are machine-dependent. Notice that the different phases are sequenced by the Call Graph and Pass Managers. The *Call Graph Manager* builds a call graph for the compilation unit and decides in which order to process each function. On the other hand, the *Pass Manager* is responsible for sequencing individual transformations and handling pre- and post-cleanup actions as needed by each pass.

5.6.3 SUIF Compiler Infrastructure

The *Stanford University Intermediate Format* (SUIF) [86–88] system is a compiler infrastructure designed to support experimental research. The compiler research community is in need for compiler infrastructures on which new technology can be implemented and evaluated. Since independently developing a new compiler infrastructure is prohibitively expensive, compiler researchers would benefit greatly from sharing investment in infrastructure development. Thus, the emphasis of the SUIF compiler system is to maximize code reuse by providing useful abstractions and frameworks for research on compiler techniques, it is powerful, modular, flexible, clearly documented and complete enough to compile large benchmark programs.

The SUIF system design is twofold. One is the *modularity* of the subsystem which allows different components to be combined easily. One or more modules developed by a programmer under this mode can be combined with a driver to produce a standalone program. A compiler can be a series of standalone programs that read and write a SUIF file, or it is a program that dynamically imports and applies a series of different modules to the program in memory. Second is the *extensibility* of the program representation which allows creating new instructions to capture new program construct semantics or new program analysis. They have predefined an object hierarchy to capture the program semantics, and users are able to further refine these abstractions for their needs. Thus, a SUIF program will always contain the same basic information but may contain different subsets of nodes represented with refined program semantics.

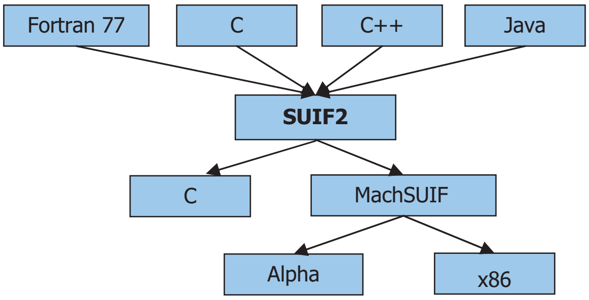


Fig. 5.19 SUIF system

As shown in Fig. 5.19, currently the SUIF system supports the front-end for four high-level languages, *i.e.*, Fortran 77, C, C++, and Java. The back-end generates three kinds of target codes, *i.e.*, C code, Alpha assembly code, and x86 assembly code. Each assembly code can be assembled by the target machine's assembler to produce a machine code. Besides, the system also provides various off-the-shelf analysis and optimization passes and relevant development tools.

The SUIF System has a simple and modular architecture which is comprised from three parts *i.e.*, kernel, modules and executable. The SUIF kernel implements a set of basic functions found to be useful across all compilation passes, a number of modules loaded dynamically under user control, and driver that controls the system operation.

The *kernel* consists of two layers, the iokernel and the suifkernel. The *iokernel* implements a persistent object system that is independent of the applications in writing compilers. Reflection is supported in the objects in this system by explicitly keeping their compositions in data structures known as Meta objects. The *suifkernel* performs three major functions and defines the intermediate representation. Firstly, this representation supports both high-level program and restructuring transforms as well as low-level analyses and optimizations. Secondly, it provides functions to access and manipulate the intermediate representation. Hiding the low-level details of the implementation makes the system easier to use and helps maintain compatibility if the representation is changed. Thirdly, it structures the interface between compiler passes. SUIF passes are separate programs that communicate via files which structure the interfaces between compiler passes. The format of this file is the same for all stages of compilation. The system supports experimentation by allowing user-defined data in annotations.

The major components in the SUIF compiler system are structured as *modules*, each of which is a C++ class identified by a unique module name. A *dynamically linked library* (dll) contains one or more modules. The dll includes an `initL<dllname>` function, invoked at the time the library is loaded, which registers all the modules within the library in the *SuifEnv*. The system comes with a number of basic modules, as well as some tools to help user construct their own modules. Modules can be a set of nodes in the intermediate representation. The infrastructure includes a set of basicnodes that represent a number of basic programming constructs, and a set of suifnodes that capture standard programming constructs in a standard language, such as C and Fortran. Users can easily define new program representations using *hoof* (a high-level specification language) or a set of nodes in a program analysis pass. The infrastructure comes with a number of basic modules such as loading and printing a SUIF program. User can easily define new passes by deriving them from the basic pass modules and supplying the specific processing functions. The *Module Subsystem*, a part of the SuifEnv, is the central repository that keeps a list of all known modules in the system. The Module Subsystem has two important functions: it registers modules when a library is loaded, and it invokes modules by passing to them all the relevant parameters.

A module can be either a set of IR nodes or a compiler pass. The *Pass* class is derived from the Module class, and the *Pipelineable Pass* is derived from the Pass class. The user can define new passes by simply sub-classing these classes and specifying only the functions to be applied to the various components in a program representation.

The standard method is to apply a pass to all the procedures in a SUIF program before applying another procedure. Pipelineable passes, however, allow the freedom to apply the passes in a pipelined fashion. That is, the driver can operate on a procedure at a time; it can apply a series of different passes on the same procedure before applying them to another procedure. Pipelining the passes improves the locality of the compiler which can be important for large codes.

To create a compiler or a standalone pass, the user needs to supply a "main" program that creates the SuifEnv, imports the relevant modules, loads a SUIF program, applies a series of transformations on the program, and eventually writes out the information.

Extensibility in SUIF is provided by an extensible class hierarchy. The SUIF compiler is built from a set of discrete executables or passes, where each pass communicates with other passes via a file system. Internally, SUIF represents a code as an object-based hierarchy of symbol tables and lists of abstract syntax trees (ASTs). Each expression tree represents one code statement encoded in a post-order tree structure. A pass communicates information to a subsequent pass in two ways: code transformation and annotations. Annotations are markers that can be attached to almost all objects in an intermediate form, instructions and symbol table entries being the most common targets. Annotations contain lists of simple data items, such as strings and integers, or SUIF objects, such as symbols and types. More complex structured annotations, containing full C structures, can also be created. These annotations can be read and re-written by a pass to obtain or expand on previously determined information.

Communication through the file system results in slower compile times, but it permits the compilation sequence to be varied and custom passes inserted with little effort.

The SUIF system characteristics make it quick and easy for a SUIF compiler writer to develop a flexible and modular compiler system and to build modules that can inter-operate with modules developed independently by other research groups. The interface of the suifdriver program that we developed is demonstration of the flexibility and modularity provided by the SUIF compiler framework. The suifdriver accepts a simple scripting language that defined compiler passes that should be applied to some SUIF programs in a file. The SUIF imports the intermediate representation (IR) formats defined in the basicnodes and suifnodes modules, which are used by the compiler pass (mypass) in a dynamic linked library called mylibrary. The library contains a registration function that the suifdriver can invoke to import the library. Once a module is registered, the suifdriver will accept the module name as additional commands in the rest of the compilation session. In the above session, the user then loads in a SUIF program generated from some other compiler passes. The user pass will work on any SUIF program, which includes information encapsulated by the basicnodes and suifnode IR. That is, the SUIF program may have been generated by using a superset of IR nodes; the user pass will still work by simply importing the pass and loading the extended SUIF program into the same environment without any recompilation.

Machine SUIF is a flexible and extensible infrastructure for constructing compiler back-ends. With it, we can readily construct and manipulate machine-level intermediate forms, and emit an assembly language, a binary object or C code. The system comes with back-ends for the Alpha and x86 architectures. One can easily add new machine targets and develop profile-driven optimizations. Though Machine SUIF is built on top of the Stanford SUIF system, the analyses and optimizations within Machine SUIF are not SUIF specific. By rewriting the implementation on an extensible interface layer, we can easily port the Machine-SUIF code base to another compilation environment. This interface is referred as an *Optimization Programming Interface* (OPI). OPI allows us to write optimizations and analyses that are parameterized with respect to both a target and an underlying environment. We use this capability to share optimization passes between Machine SUIF for dynamic code optimization.

Machine SUIF is designed with three primary goals in mind. First and foremost, Machine SUIF has to be easy to use, especially if we want to do something simple, and straightforward to retarget such that we could use it in architectural investigations. Second, it has to support the modular development of sophisticated optimizations. The idea is to allow the ability to contribute passes and benefit from the efforts of others. Finally, it has to be built in a manner that permits reuse of existing optimizations directly in an optimization environment with significantly different constraints.

Machine SUIF is aimed at providing a framework for machine-dependent optimizations. It provides an intermediate representation related to SUIF, but is aimed at capturing low-level machine details rather than high-level program constructs. Machine SUIF like SUIF can be partitioned into two sections: the support library and a set of passes that make use of the library. At the present time, the only passes available with the machine SUIF distribution are code generators which map the SUIF intermediate representation of a code to the machine SUIF representation, and a machine language printer for translating the machine SUIF intermediate form to ASCII assembly language less suitable for the target machine's native assembler. Any number of optimization and scheduling passes could be inserted between code generation and ASCII generation.

Translation of the SUIF representation of a code into a machine SUIF representation is straightforward. SUIF represents a code as a list of expression trees with the nodes of each expression tree made up of symbol table entries and RISC like instructions. Since previous SUIF passes have propagated all information needed for code generation to the nodes at which that information is needed, the appropriate instruction sequence is determined solely by a recursive post-order decent down each expression tree.

5.6.4 IMPACT Compiler Infrastructure

In 2002, the University of Illinois released the IMPACT (*Illinois Microarchitecture Project utilizing Advanced Compiler Technology*) compiler into the open-source domain as part of GELATO [89].

The IMPACT-I C compiler provides a platform for studying new code optimization techniques for multiple-instruction-issue architectures [90]. It performs several code transformations that enlarge the scope of static scheduling, including function inline expansion, instruction placement, loop unrolling, loop peeling, and branch expansion. The compiler also performs several code transformations that reduce the depth of critical paths, including induction variable expansion, register renaming, global variable register allocation, operation combining, operation folding, and memory disambiguation. Using a profiler, one can measure the execution count of every operation and collect branch statistics. Compile-time decisions are based on a composite of 20 profiles. It derive the best and the worst case execution time of each superblock, assuming ideal cache. The worst case is due to long operation latencies that protrude from one superblock to another superblock.

OpenIMPACT [91] moves IMPACT from research to general use. To allow each source file be compiled standalone as a general compiler does, OpenIMPACT embeds IR in object files and libraries for further global optimization. Due to its

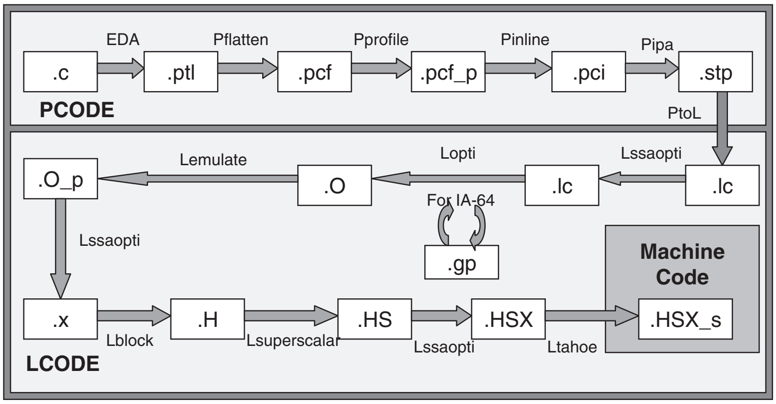


Fig. 5.20 IMPACT compiler flow

heritage as a research compiler, OpenIMPACT is designed to achieve maximal output code performance with little concern for compiler time and compiler memory usage (Fig. 5.20).

OpenIMPACT encompasses many of the advanced compilation techniques developed by the IMPACT research team, including

(1) programmatic logic analysis,
(2) predicated compilation,
(3) interprocedural pointer analysis,
(4) instruction-level parallelism optimizations,
(5) profile-based optimization, and
(6) speculative hyperblock acyclic and modulo scheduling.

Chapter 6
Implementation of H.264 on PLX

In this chapter, we will demonstrate our experiences in implementing H.264 encoder on a PLX-based platform from system level design, virtual prototyping, to ASIP design.

6.1 Instruction Set Decision for H.264

Most computation of H.264 is spent on Motion Estimation as described in Section 2.3.2. Its kernel operation is a Sum of Absolute Differences (SAD):

$$SAD\,(m,n)=\sum_{i=0}^{N-1}\sum_{j=0}^{N-1}|\,C(i,j)-R(i+m,j+n)|$$

A 16×16 macroblock contains 256 pixels, and each video pixel is stored as an 8-bit data. Since the SAD computation requires a 16-bit resolution, the 8-bit pixel value has to be expanded into a 16-bit one before performing subtraction. Using a 128-bit SWP-SIMD ALU, we can compute eight pixels at a time.

But SAD is a losable operation, we only wish to know which SAD is minimum in search range. Thus, its resolution if reduced into 8-bit is still working, and its computation speed will be doubled.

The 16×16 macroblock is composed of sixteen 4×4 sub-blocks as shown in Fig. 2.15. If the pixel value is right shifted into a 4-bit resolution, the SAD of the 4×4 sub-blocks can remain in an 8-bit resolution. When the pixel values vary in a very large range, their lower 4 bits are ignorable. In such case, this 4-bit shift solution becomes a feasible one.

Another solution is using saturation operation as described in Section 4.1. While our goal is to find the minimum one, if the minimal SAD never exceeds 254, saturating a large SAD value into 255 will not destroy the selection.

Table 6.1 lists the PSNR of reconstructed frames using the above mentioned 3 resolution methods on two sequences. The results are obtained by running simulation on our PLX-H.264 virtual prototype platform (as shown in Fig. 6.2). As shown in the table, the sequence *Stefan* has fast moving, and sequence *Weather* has

Table 6.1 PSNR in SAD with 3 different resolutions

PSNR	Stefan			Weather		
Frame	16bit	8bit-sat	4bit-shift	16bit	8bit-sat	4bit-shift
1	37.35942	37.35031	37.20614	37.52711	37.52356	37.44326
2	37.29617	37.28674	37.03496	37.71939	37.71754	37.59165
3	37.25709	37.23658	36.94312	37.80259	37.80131	37.65876
4	37.07474	37.07342	36.72818	37.81671	37.81664	37.65552
5	36.99427	36.99151	36.60071	37.81785	37.81560	37.66196
6	36.91317	36.91058	36.50841	37.79967	37.79784	37.63698
7	36.92029	36.88299	36.45134	37.76962	37.76885	37.61353
8	36.88774	36.87031	36.35291	37.74985	37.74801	37.57269
9	36.81931	36.80784	36.33460	37.73409	37.72860	37.53180
Average	37.05802	37.04559	36.68449	37.74854	37.74644	37.59624

little moving. From the results shown on the two sequences, the 8-bit saturation method is only a little bit worse than the 16-bit method. As revealed by the simulation results, we implemented the 8-bit saturation operation into our PLX processor instruction set.

6.2 Hardware/Software Partitioning

After the instruction set is defined, we can evaluate the performance upper bound of some kernel functions by assuming that every instruction takes only one cycle without considering the memory latency. Some kernel functions in H.264, such as Motion Estimation, Transformation, Quantization, De-block filter and Entropy Encoding, have been described in Section 2.3. While a parallel compiler is not ready, these algorithms are coded in assembly language for performance simulation.

Using a 128-bit SWP-SIMD processor, a row (16 pixels) of macroblock elements can be loaded into a register at once. Utilizing the SWP-SIMD feature, the sixteen 4×4 sub-blocks will be computed in an order as shown in Fig. 6.1, which operates on four 4×4 sub-block elements in parallel. The matrix transpose operations as described in Section 5.2.5 are used to re-organize the data flow in the SWP-SIMD PLX core. All the 41 motion vector variations can be composed by the obtained 16 results. The PLX assembly code was shown in Fig. 3.5, which takes 273 cycles for calculating the SADs for a given displacement (m, n).

Using a 3-step search method with a search range of 16, we need 6825 cycles to find the best motion vector for a macroblock. If the processor works at 100MHz, it spends 0.6% of the computation resources for a 176×144 QCIF image if only one reference frame is used. If the image size is 1024×768 and uses 5 reference frames, the motion estimation operations will occupy 100% of the computation resource.

The remaining kernel functions in total only take 7700 cycles, or 23% of the computation resources for a 1024×768 image. Most of them are spent on entropy encoding because it is unable to parallelize in the SWP-SIMD PLX core.

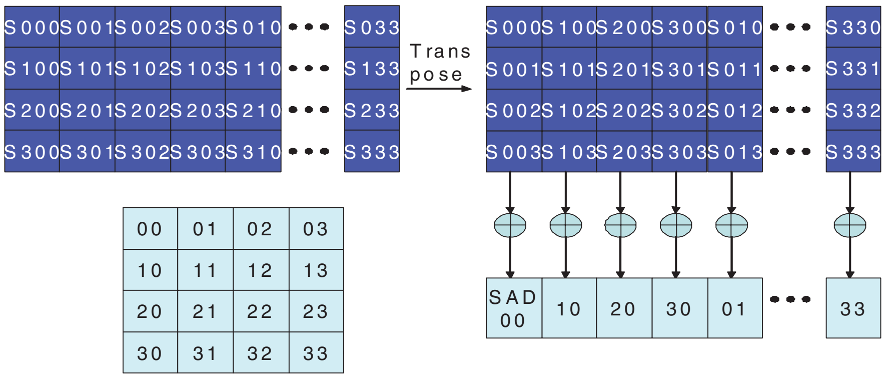

S000 S001 S002 S003 S010 ••• S033
S100 S101 S102 S103 S110 ••• S133
S200 S201 S202 S203 S210 ••• S233
S300 S301 S302 S303 S310 ••• S333
Trans pose
S000 S100 S200 S300 S010 ••• S330
S001 S101 S201 S301 S011 ••• S331
S002 S102 S202 S302 S012 ••• S332
S003 S103 S203 S303 S013 ••• S333
00 01 02 03
10 11 12 13
20 21 22 23
30 31 32 33
SAD 00 10 20 30 01 ••• 33

Fig. 6.1 SAD calculation in a 128-bit SWP-SIMD

By simple evaluation, we know that H.264 encoder can be implemented on a single PLX processor if the required image size is small. But for higher image quality, another core dedicated for motion estimation computing is required. It could be an ASIC or a dual-core PLX.

Energy consumption is the most critical constraint for a portable consumer electronic device. Instead of increasing clock frequency, improving parallelism with a little area overhead is more power efficient. Since H.264 is a memory-dominant application, simultaneous multi-threading (SMT) is good to hide memory latency. Entropy encoding and many non-kernel functions cannot fully utilize the SWP-SIMD feature, because PLX integrates scalar and vector operations in a single core; thus performing a 32-bit scalar function using 128-bit registers is a waste. VLIW is an alternative way to reduce such kind of waste.

6.3 Untimed Virtual Prototype

After the PLX *instruction set architecture* (ISA) is decided, we wish to involve into hardware and software design detail as soon as possible. Before the PLX processor design is completed, a hardware prototype served as a software development platform as described in Section 2.1 is not available yet. *Virtual prototype* offers the ability for developing software in an early stage. A *virtual prototype* means that it is not a real target design, but it can simulate the behavior of the target design using a higher abstraction level TLM modeling method, as described in Sections 3.1 and 3.3. A untimed TLM modeling that provides the architecture designers with an PLX *instruction set simulator* (ISS) for them to focus on the *Programmer View* (PV) of a design will be presented in this section. Also a timed TLM modeling concerning on the PV plus Timing (PVT) view on the peripheral I/O devices that can be used to model system input/output functions will be depicted in next section.

To design a compiler which can automatically optimize code for SWP-SIMD, VLIW and multi-threading is still a challenge. The first PLX compiler is implemented by LCC infrastructure as described in Section 5.6.1, which implements PLX instruction set into LCC backend without considering any of the DLP, ILP and TLP optimizations. Software team can use it to develop RTOS and user interface. A prototype of such kind of OS development will be described in Chapter 7.

In designing a virtual prototype platform, an important concern is able to refine the design under development. Eventually, the software developed on this virtual prototype platform should be able to port to real product directly. Thus, we need to emulate peripheral operations on the system.

In order to emulate the H.264 encoder operations, we need four basic peripheral devices: a camera to capture an image, an LCD to display the image, a Storage to store the recorded video, and a Communication interface to transfer the image to other devices. The system architecture of our virtual prototype platform is shown in Fig. 6.2.

In most products, Camera and LCD are implemented on a shared memory architecture. The Camera module gets pixels from CMOS image sensor in a fixed rate and stores the image in a pre-defined memory location. The LCD module reads pixel values from the pre-defined memory location and converts them into a format compliant to the LCD control protocol. In our virtual platform, the memory space that PLX can access is mapping to a memory array during simulator initialization. In the simulator, we created two threads to emulate the above behavior. The *Camera_Thread* gets an image from a system interface and puts the image data into the pre-defined memory location. Windows handle the Camera device by a *DirectShow* filter, which can be accessed by an application through a system defined interface (`IBaseFilter` in following sample code). After that, the *LCD_Thread* gets the bitmap stored in this memory location and put them into a window device context (`hdc` in the following `LCD_Thread` code), then, our virtual prototype platform will display the context on a window. These two threads are woken up by a system timer to emulate the fixed-frame-rate behavior.

There are many communication physical layers available in the world, including Ethernet, wireless-LAN or IEEE-1394. Each has its own protocol. To implement a communication protocol needs a lot of man-power. At the early design stage, it is better to implement the communication protocol using a message-passing API. A unified `Send` and `Receive` API was defined, and OS developers can refine this

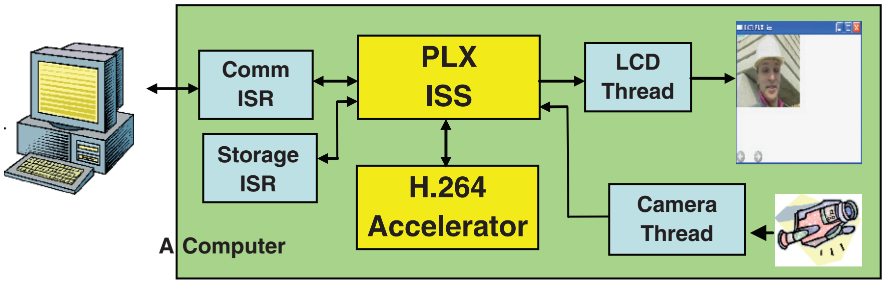


Fig. 6.2 A PLX-H.264 virtual prototype platform

implementation to satisfy their need. We implemented this message-passing API on an interrupt service routine (ISR). When software needs to `Send` or `Receive` a message, its data buffer location and length are put in respective registers and the associated ISR is called by a *trap* instruction with an interrupt vector. In our virtual prototype, the data buffer location and length registers are re-directed to a system network socket.

The storage interface is also implemented by ISR. The `FileRead` and `FileWrite` APIs respectively get and put a data with its buffer location and length stored in registers, and call a corresponding ISR by invoking the *trap* instruction. In our virtual prototype implementation, these registers are re-directed to a `FileRead` and a `FileWrite` system calls.

The virtual H.264 accelerator `ME_PROC` is also implemented by ISR. This major 1-D motion estimation component in the H.264 virtual accelerator hardware is written according to the real hardware design as shown in Fig. 2.16. It computes all the 41 motion vectors for different macroblock combinations in an image. PLX is in charge to send information on the current frame location, reference frame location, motion vector location and image size to the ME accelerator through an on-chip bus. After the accelerator is started, it becomes the second bus master which will occupy bus bandwidth to load pixels from memory. In our virtual prototype implementation, the ISR is re-directed to a C code subroutine `ME_PROC`, and the bus hand-shaking is left to the timed TLM model that will be presented in next section.

The SystemC code of this untimed virtual prototype is listed as follows.

```
typedef union   {
  unsigned char    b[16];
  unsigned short   w[8];
  unsigned long    d[4];
  unsigned __int64  l[2];
} B128;
class PLX   {
public:
  B128             R[32];
  bool             predflag[8];
  int              PC;
  bool             halt;
  unsigned char    *ram;
  BITMAPINFO       bm;
  ISampleCaptureGraphBuilder *pCapture;
  IGraphBuilder    *pFg;
  IBaseFilter      *pVideo;
  IAMStreamConfig  *pVSC;
  void            init(int argc, char *argv[]);
  void            ALU(void);
}  plx;

void main(int argc, char *argv[])
{
  MSG msg;
  plx.bm.bmiHeader.biSize=sizeof(BITMAPINFOHEADER);
  plx.bm.bmiHeader.biWidth=352;
  plx.bm.bmiHeader.biHeight=-288;
  InitThread(LCD_Proc);
  InitCamera();
  InitThread(Camera_Proc);
  plx.init(argc, argv); //Load code and testge into RAM
  do {
    plx.ALU();
  } while(!plx.halt);
 }

 void PLX::init(int argc, char *argv[])
 {
    memset(R,0,sizeof(R));
    memset(predflag,0,sizeof(predflag));
    predflag[0]=1;
    ram=new unsigned char[16384*1024]; //16MB ram
 }

void PLX::ALU(void)
{
    unsigned long op=*(unsigned long *)(ram+PC+CODEstart);
    unsigned char p=op>>29;
    unsigned char opc=(op>>23)&0x3F;
    unsigned char Rd=(op>>18)&0x1F;
    unsigned char Rs1=(op>>13)&0x1F;
    unsigned char Rs2=(op>>8)&0x1F;
    int i;
    DWORD nn;
    if (!predflag[p]) return;
    switch(opc) {
    case HALT: halt=1; break;
    case IDLE: context_switch(); break;
    case JMP: PC=op&0x7FFFFF; break;
    case MREAD: R[Rd].l[0]=*(__int64 *)(ram+R[Rs1].d[0]);
    case MWRITEB: *(char *)(ram+R[Rs1].d[0])=R[Rd].b[0];
    case MWRITEW: *(short *)(ram+R[Rs1].d[0])=R[Rd].w[0];
    case MWRITED: *(long *)(ram+R[Rs1].d[0])=R[Rd].d[0];
    case MWRITEL: *(__int64 *)(ram+R[Rs1].d[0])=R[Rd].l[0];
    case PADD:
      switch(op&3){
      case 0: for(i=0;i<16;i++)
                R[Rd].b[i]=R[Rs1].b[i]+R[Rs2].b[i]; break;
      case 1: for(i=0;i<8;i++)
                R[Rd].w[i]=R[Rs1].w[i]+R[Rs2].w[i]; break;
      case 2: for(i=0;i<4;i++)
                R[Rd].d[i]=R[Rs1].d[i]+R[Rs2].d[i]; break;
      }
    case TRAP:
      switch(op&0xFFFF) {
      case COMM_ISR:
        switch(R[1].w[0]) {
       case C_OPEN:R[2].d[0]=socket(AF_INET,SOCK_STREAM,0);
                    bind(R[2].d[0], R[3].d[0]);
                    break;
       case C_RECV: recv(R[2].d[0],ram+R[3].d[0],
                    R[4].d[0]); break;
       case C_SEND: send(R[2].d[0],ram+R[3].d[0],
                    R[4].d[0]); break;
         }
         break;
       case STORAGE_ISR:
         switch(R[1].w[0]) {
         case F_OPEN: R[2].d[0]=(DWORD)fopen((char *)ram+
                    R[3].d[0], (char *)ram+R[4].d[0]); break;
         case F_READ: fread(ram+R[3].d[0],R[4].d[0],1,
                       (FILE *)R[2].d[0]); break;
         case F_WRITE: fwrite(ram+R[3].d[0],R[4].d[0],1,
                        (FILE *)R[2].d[0]); break;
         }
       case ME_ISR:    ME_PROC(ram+R[3].d[0],ram+R[4].d[0],
                   ram+R[5].d[0], R[6].w[0],R[6].w[1]);
                   break;
       }
    }
}

void ME_PROC(unsigned char *cur, unsigned char *ref,
             unsigned char *mv, short wx, short wy)
{
    for(int y=0;y<wy;y+=16) {
      for(int x=0;x<wx;x+=16) {
        ... //compute all MV
      }
    }
}

LRESULT CALLBACK LCD_Thread(HWND hWnd, UINT message)
{
  PAINTSTRUCT ps;
  HDC hdc;
  switch (message)
  {
   case WM_TIMER:
     if (wParam==Timer_ID) {
         hdc=GetDC(hWnd);
```

```
          SetDIBitsToDevice(hdc,0,0,352,288,0,0,0,288,
             plx.ram+LCDstart,&plx.bm,DIB_RGB_COLORS);
          ReleaseDC(hWnd,hdc);
      }
      return 0;
    default:
      return DefWindowProc(hWnd, message, wParam, lParam);
   }
 }

 void InitCamera()
 {
   CoCreateInstance(CLSID_FilterGraph, NULL, CLSCTX_INPROC,
                             IID_IGraphBuilder, &plx.pFg);
   BindToObject(0, 0, IID_IBaseFilter, &plx.pVideo);
   plx.pFg->AddFilter(plx.pVideo);
   plx.pCapture->SetFiltergraph(plx.pFg);
 }

 LRESULT CALLBACK Camera_Thread(HWND hWnd, UINT message)
 {
  PAINTSTRUCT ps;
  HDC hdc;
  AM_MEDIA_TYPE *pmt;
  switch (message)
  {
    case WM_TIMER:
      plx.pCapture->FindInterface(&PIN_CATEGORY_CAPTURE,
                             &MEDIATYPE_Video, plx.pVideo,
                          IID_IAMStreamConfig, &plx.pVSC);
      plx.pVSC->GetFormat(&pmt);
      memcpy(ram+CameraStart,(BITMAPINFO)(pmt-> pbFormat));
      return 0;
    default:
      return DefWindowProc(hWnd, message, wParam, lParam);
   }
}
```

6.4 Timed SystemC Modeling

In above untimed TLM implementation, the PLX processor is idle while ME_PROC is executing. In a multi-threading processor, OS will issue another thread when one thread is idle. At such circumstance, H.264 accelerator will be a bus master to access memory.

Another important part is on cache performance. H.264 is memory-critical application, where memory bandwidth dominates system performance. A system level model considering memory interface detail can help hardware designers to design a cache system in a more accurate way.

Thus, we need to upgrade the simulator to an Approximately Timed TLM level (*i.e.*, PV plus Timing view) using SystemC. In this level of abstraction, the memory read/write instructions and bus timing are cycle accurate, other instructions are ideal single cycle execution, and cache replacement strategy is also cycle accurate to exactly reflect memory latency.

In the `H264` module `ME_PROC`, motion vectors (MEs) are computed and saved into RAM when this function is called. Memory request is issued in a pre-defined rate to emulate the bus bandwidth competition, but the detailed memory access information is not used in running ME simulation to reduce the simulator design complexity.

The SystemC code of this timed TLM model is listed as follows.

```
void sc_main(int argc, char *argv[])
{
    sc_clock  clk("CLOCK",clockcycle,0.5,1);
    PLX plx("PLX");
    plx.clk(clk);
    plx.init(argc, argv); //Load code and testge into RAM
    do {
         sc_start(clk,clockcycle);
    } while(!plx.halt);
}

SC_MODULE(PLX)
{
    sc_in_clk      clk;
    sc_signal<sc_uint<32> >   rom_addr, rom_rdata;
    sc_signal<sc_uint<1> >    rom_req, rom_ready;
    sc_signal<sc_uint<32> >   ram_addr, ram_size;
    sc_signal<sc_uint<64> >   ram_wdata, ram_rdata;
    sc_signal<sc_uint<1> >    ram_req, ram_ready;
    sc_signal<sc_uint<32> >   ic_addr, ic_rdata;
    sc_signal<sc_uint<1> >    ic_req, ic_ready;
    sc_signal<sc_uint<32> >   dc_addr, dc_size;
    sc_signal<sc_uint<64> >   dc_wdata, dc_rdata;
    sc_signal<sc_uint<1> >    dc_req, dc_wrire, dc_ready;
    sc_signal<sc_uint<32> >   h264_addr, h264_size;
    sc_signal<sc_uint<64> >   h264_wdata, h264_rdata;
    sc_signal<sc_uint<1> >    h264_req, h264_wrire, h264_ready;
    sc_signal<sc_uint<8> >    intreq;
    B128            R[32];
    bool            predflag[8];
    int             PC;
    bool            halt;
    unsigned char   *ram;
    BITMAPINFO      bm;
    int             icachests;
    int             dcachests;
    int             h264sts;
    int             arbitersts;
    unsigned __int64   dcacheram[ROWS][COLS];
    unsigned        dcachetag[ROWS];
    bool            dcachedirty[ROWS][COLS];
    unsigned long   icacheram[ROWS][COLS*2];
    unsigned         icachetag[ROWS];
    void             init(int argc, char *argv[]);
    void             ALU(void);
    void             ICACHE();
    void             DCACHE();
    void             H264();
    void             ARBITER();

    SC_CTOR(TOP)
    {
      plx->clk(clk);
      icache->clk(clk);
      ...
      icachests=IDLE;
      dcachests=IDLE;
      h264sts=IDLE;
      arbitersts=IDLE;
    }
}

void   PLX::DCACHE()
{
    switch(dcachests) {
    case IDLE:
     if (ram_req.read()) {
           if (hit(ram_addr.read(),&row,&col))
             dcachests.write(READY);
           else if (alldirty()) dcachests.write(SELDIRTY);
           else dcachests.write(MISS);
     }
     break;
    case READY:
      if (ram_write.read()) {
        dcacheram[row][col]=ram_wdata.read();
        dcachedirty[row][col]=true;
     } else ram_rdata.write(dcacheram[row][col]);
      if (ram_req.read()) {
          if (hit(ram_addr.read(),&row,&col))
            dcachests.write(READY);
          else if (alldirty()) dcachests.write(SELDIRTY);
          else {row=LRU_nondirty(); dcachests.write(MISS);}
     }
     break;
    case SELDIRTY:
      row=LRU_dirty();
      cols=0;
      dcachetag[row]=rom_addr.read()/COLS/WORDS;
      dcachests.write(WRITEBACK);
      break;
    case WRITEBACK:
      dc_req.write(1);
      dc_addr.write(dcachetag[row]*COLS*WORDS);
      dc_write.write(1);
      dcachests.write(WAITW);
      break;
    case WAITW:
      if (dc_ready.read()) {
        dc_wdata.write(dcacheram[row][col]);
        dcachedirty[row][col]=0;
        col++;
        dc_addr.write((dcachetag[row]*COLS+col)*WORDS);
        if (col==COLS) {
          if (ram_write.read()) dcachests.write(READY);
          else dcachests.write(MISS);
        }
      }
       break;
       case MISS:
        dc_req.write(1);
        dc_addr.write(dcachetag[row]*COLS*WORDS);
            dc_write.write(0);
            dcachests.write(WAITM);
            break;
        case WAITM:
            if (dc_ready.read()) {
               dcacheram[row][col]=dc_rdata.read();
                col++;
                if (col==COLS) dcachests.write(READY);
                else dc_addr.write((dcachetag[row]*COLS+col)*WORDS);
            }
           break;
         }
}

void     PLX::H264()
{
        switch(h264sts) {
        case IDLE:
            if (ram_req.read()&&ram_write.read()) {
               switch(ram_addr.read()) {
                case REG_CUR: cur.write(ram_wdata.read());
                case REG_REF: ref.write(ram_wdata.read());
                case REG_REF: mv.write(ram_wdata.read());
                case REG_WX:  wx.write(ram_wdata.read());
                case REG_WY:  wy.write(ram_wdata.read());
                case   REG_CTRL: if (ram_wdata.read()&1) {
                                     ME_PROC(cur,ref,mvs,wx,wy);
                                     for(i=0;i<N;i++) ram[mv+i]=mvs[i];
                                     cnt=0;
                                     h264sts.write(RUN);
                                }
                                 break;
             }
             break;
        case RUN:
            if (cnt&DORAM) {
              h264_req.write(1);
              h264_addr.write(ref.read())

              h264sts.write(WAIT);
           }
           if (++cnt >= MAXCNT) {
             intreq.write(ISR_H264END);
             h264sts.write(IDLE);
           }
            break;
         case WAIT:
             if (h264_ready.read()) h264sts.write(RUN);
}
```

```
void PLX::ALU(void)
{
         if (ram_req.read()) {
           if (ram_ready.read()) {
               if (!ram_write.read()) R[readRd]=ram_rdata.read();
                ram_req.write(0);
            } else return;
         } else if (intreq.read()!=0) { //Check interrupt
            op=(TRAP<<23) | (intreq.read());
            intreq.write(0);
        } else if (rom_ready.read()) {
            op=rom_rdata.read();
            PC=PC+4;
        } else return;
    unsigned char p=op>>29;
    unsigned char opc=(op>>23)&0x3F;
    unsigned char Rd=(op>>18)&0x1F;
    unsigned char Rs1=(op>>13)&0x1F;
    unsigned char Rs2=(op>>8)&0x1F;
    if (!predflag[p]) return;
    switch(opc) {
    case MREAD: ram_req.write(1);
                ram_write.write(0);
                ram_addr.write(R[Rs1].d[0]);
                readRd=Rd;
                break;
    case WRITEB: ram_req.write(1);
                 ram_write.write(1);
                 ram_addr.write(R[Rs1].d[0]);
                 ram_wdata.write(R[Rd].l[0]);
                 break;
    case TRAP:
      switch(op&0xFFFF) {
      case COMM_ISR:
        break;
      case STORAGE_ISR:
        switch(R[1].w[0]) {
        case F_OPEN: R[2].d[0]=(DWORD)fopen((char *)ram+
                     R[3].d[0], (char *)ram+R[4].d[0]); break;
        case F_READ: fread(ram+R[3].d[0],R[4].d[0],1,
                       (FILE *)R[2].d[0]); break;
       case F_WRITE: fwrite(ram+R[3].d[0],R[4].d[0],1,
                      (FILE *)R[2].d[0]); break;
      }
     case ME_ISR:  ME_PROC(ram+R[3].d[0],ram+R[4].d[0],
                 ram+R[5].d[0], R[6].w[0],R[6].w[1]);
                 break;
     }
  }
}

void PLX::Arbiter(void) {
       switch(grant) {
       case 0:
         if (dc_req) grant=GRANT_DC;
         else if (ic_req) grant=GRANT_IC;
         else if (h264_req) grant=GRANT_H264);
          break;
        case GRANT_DC:
          if (!dc_req|| timeout) {grant=0; dc_ready=0;}
```

```
            else {dc_rdata=*(__int64 *)&ram[dc_addr];
            dc_ready=1;}
            break;
          ...
}
```

6.5 PLX Chip Design

The PLX processor is designed to run at 100 MHz in 0.18 μm TSMC process. A Wallace-tree 32-bit multiplier is designed to work on 100 MHz without extra pipelining buffer. Since all instructions have to be executed in one cycle, a standard RISC architecture is adopted.

The PLX processor design is shown in Fig. 6.3, which owns a 6-stage pipeline RISC architecture. The IFETCH unit contains two instruction buffers (IBUFF) for the two threads to work simultaneously. The DECODE unit decodes the last two instructions stored in IBUFFs. The ISSUE unit selects these two decoded instructions, calculates their operand addresses, and generates the ALU control signals. The OPERAND unit fetches operands from the register file that contains 64 128-bit registers. The Control Program Status Register (CPSR) inside the register file contains a system timer, an interrupt return address, a cache invalidation control, and issue mode control flags. The execution unit contains ALU and Load/Store unit. The Load/Store unit sends memory request to DCACHE or other on-chip IP through the Advance High-performance Bus (AHB).

The 128-bit ALU is composed of four 32-bit datapath-adjustable ALUs and a shuffle unit for the inter-subword permutation operations. It is able to configure as SIMD or VLIW mode by setting the two control bits in the instruction code.

In the SIMD mode, one instruction is fetched from the IBUFF in every cycle, and duplicated into the four 32-bit datapath-adjustable ALUs. In the VLIW mode, the

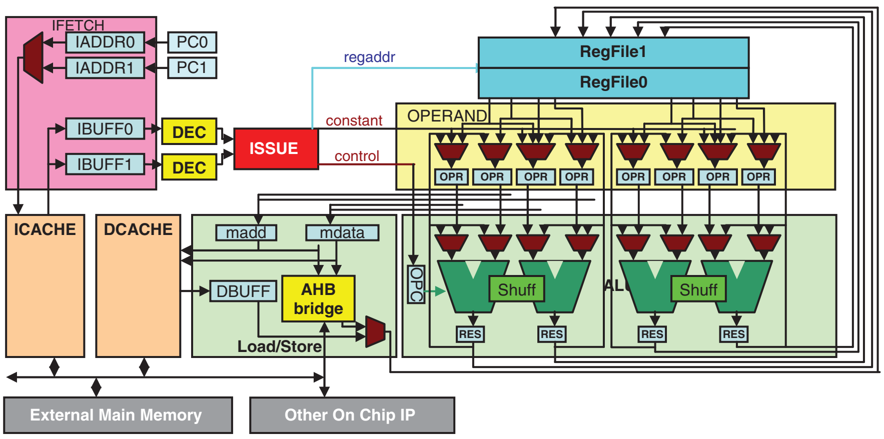


Fig. 6.3 PLX chip architecture

ISSUE unit fetches four decoded instructions from the IBUFF and dispatches them into the four 32-bit datapath-adjustable ALUs.

Two bits in the PLX instruction word are allocated to support VLIW. The SCALAR bit indicates that the instruction requires only a 32-bit ALU. The ILP bit indicates that the instruction is to be executed in parallel with a previous instruction. Only the instructions with both SCALAR and ILP bits on are combined into a VLIW instruction by the ISSUE unit.

When both threads are enabled by the OS, the ISSUE stage fetches the codes from the two IBUFF units in an interleaving way. Each thread can only access the thirty-two 128-bit registers. In such a way, the designed PLX processor can be used to support simultaneous multi-threading (SMT).

Chapter 7
Real-Time Operating System for PLX

PLX, as described in Sections 4.2.2 and 6.5, is a processor with wordsize scalable, fully subword-parallel (native SIMD) instruction set architecture that supports high-performance, low-cost multimedia information processing, 3-D graphical processing and permutation instructions for security operation. Several development tools, such as the parallel compiler mentioned in Chapter 5, the assembler, and the *instruction set simulator* (ISS), are available to help users in developing their applications on the PLX virtual platform. However, it lacks an operating system (OS).

Applications, such as wireless security or multimedia, are often quite complicated to design and implement. Without an OS, application designers have to handle and synchronize the data communication among tasks and take care of hardware resource conflicts. This may lead to inefficient designs and error-prone implementations as application designers cannot focus on the main functionality of their applications. An operating system is thus required to support application designers in efficiently and conveniently using the PLX processor to accelerate their applications.

In this chapter, we will illustrate how we developed, from scratch, a *real-time operating system* (RTOS) for the PLX processor. The main goal of the RTOS is to design and implement a free, open-source, working and efficient RTOS, targeted specifically for native SIMD ISA configurable processors, such as the PLX. The RTOS also considers performance, power, and portability issues.

The RTOS consists of several components as shown in Fig. 7.1 that cooperate with each other to execute application tasks. We will now describe each component of the RTOS. The first component, also the most important one, is the *scheduler* which is capable of multiprogram scheduling. Multiple tasks can be scheduled to efficiently use the PLX processor such that when a task is doing I/O, other tasks can be scheduled to run on the RTOS. We designed a simple scheduler as described in Section 7.1.

The second component is the *dispatcher*, which accepts requests from the scheduler and performs context switch between the current task and the next task as decided by the scheduler.

The third component is a *memory manager*, which is capable of memory placement, memory allocation and deallocation, and shared memory management for communication among tasks. Besides, the RTOS also supports the use of *memory-mapped* I/O (MMIO) method between the CPU and peripheral devices.

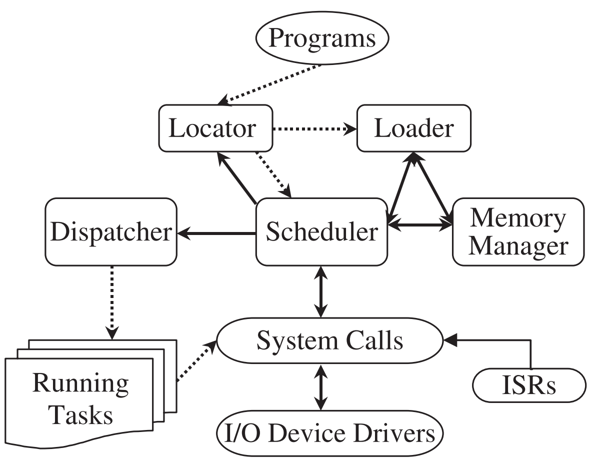


Fig. 7.1 RTOS components and relations

The fourth component comprises a *locator* and a *loader*. The locator is responsible for address translation from logical to physical, that is, it calculates the physical address for the global or static variables. The translated physical addresses constitute the input for the loader. The loader is in charge of reading a code from external RAM and initializing the code execution environment by executing the startup code. The initialization includes assigning values to the static or global variables and determining the start addresses of the stack and the heap.

The fifth component is the *System Calls*, which include the communication and synchronization primitives, *e.g.*, the semaphores in a shared memory, which are used for communicating among tasks and for protecting the critical regions or shared data, respectively.

The last component includes two specific system calls: *interrupt service routines* (ISRs) and *I/O device drivers*. Currently, we have implemented a key ISR and a timer ISR in the RTOS. The key ISR is for inserting a new task, and the timer ISR is for counting down the time quantum for the current task. The other components, including the system calls other than memory management and I/O device drivers, will be implemented in the future.

In this chapter, we will describe in details the design and implementation of an RTOS for PLX. The remaining sections are organized as follows. In Section 7.1, we discuss the scheduler. In Section 7.2, we demonstrate the memory placement to see where the kernel, tasks, and memory-mapped I/O are arranged, and several system calls for memory allocation and deallocation are also detailed in this section. The system calls for shared memory which are based on the memory allocation are also described in this section. The content of Section 7.3 includes the implementation of communication and synchronization primitives. In Section 7.4, we introduce how our RTOS can address several issues such as real-time constraints, in multimedia applications. In Section 7.5, we introduce the toolchain that can help users to develop their applications on our RTOS for PLX. Section 7.6 contains the experimental results.

7.1 PRRP Scheduler

The design of a scheduler for PLX has gone through different implementation stages. From the beginning, we used a simple scheduler called *Single Stack Tasker* (SST) [92], which was developed in C language; however, this scheduler does not support multiprogramming and was only used to check if our toolchain flow is feasible. This scheduler was designed for general-purpose non real-time applications. In the second version, we implemented a *Priority-based, Round-Robin, and Preemptive* (PRRP) scheduler. The main goal of the new scheduler was to support not only multiprogramming, but also time-sharing. With the capability of multiprogramming, the real-time operating system can increase the utilization of the processor, and the integration of time-sharing among tasks reduces the response time for each task.

Before we go into the details on the scheduler, we first roughly define the term, *task*. A task comprises user code, data, heap, and stack pointers that form a *context*. These context data are all stored in specific sections of the main memory. We partition the main memory into several regions comprising the kernel and the user tasks. The memory layout of the kernel and the user tasks will be detailed in Section 7.2.

Several data structures are also defined and implemented in the kernel for the scheduler to control the execution flow of the tasks. As shown in Fig. 7.2, a *Task Index List* (TIL) is an array of pointers each of which points to the start address of a task structure in the main memory. A *task structure* contains the private data, such as register values before being suspended, priority value, and some flags of a task. A *Current Task Index* (CTI) data structure is used to store the address of a currently running task as its name shows. The *Round-Robin Index List* (RRIL) is a list of pointers each of which points to the physical start address of the highest priority tasks. This list can be seen as a *round-robin* (RR) queue. When the RR timer times out, we directly shift the CTI to the next entry in the RRIL, where the tasks with the same priority are equally served.

The scheduling algorithm is shown in Fig. 7.3. In the beginning, we initialize the current priority to the minimum one in our system, and the current task index

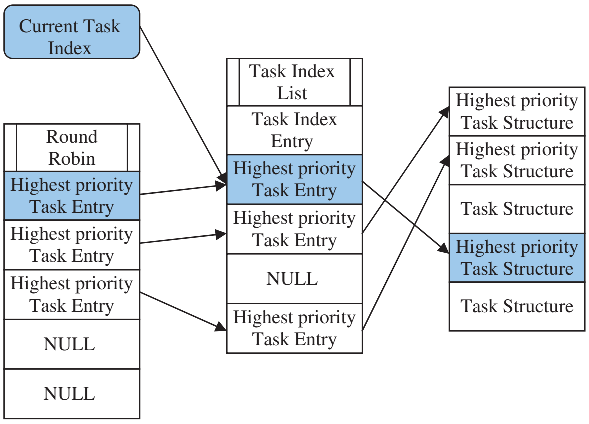

Current Task Index
Round Robin
Highest priority Task Entry
Highest priority Task Entry
Highest priority Task Entry
NULL
NULL
Task Index List
Task Index Entry
Highest priority Task Entry
Highest priority Task Entry
NULL
Highest priority Task Entry
Highest priority Task Structure
Highest priority Task Structure
Task Structure
Highest priority Task Structure
Task Structure

Fig. 7.2 Extended data structure for the PRRP scheduler

```
begin
cur_priority ← MIN_PRIORITY ;
cur_index ← -1;
i = 0;
foreach i < MAX_TASK do
  if TIL[i] = NULL then
    continue;
  endif
  if cur_priority < Priority(TIL[i]) then
    cur_index ← i;
    cur_priority ← Priority(TIL[i]);
  endif
  i++;
endfor
if cur_index =  -1 then
  RunIdletask();
else
  if TIL[cur_index] = Current_Task_Index then
    Continue_Current_Task();
  else
    Context_Switch(Current_Task_Index, TIL[cur_index]);
    Current_Task_Index ← TIL[cur_index];
  endif
endif
end
```

Fig. 7.3 PRRP scheduling algorithm in RTOS for PLX

is assigned to −1. In the `for` loop, we search on the TIL array and see if there is any task with higher priority than the current task. If a task with a higher priority exists, we set the current priority to the higher one and exit the `for` loop. After leaving the `for` loop, we immediately check whether the value of current index is still −1; if it is, we run the idle task which is a kernel task and contains an infinite loop performing some dummy instructions. If the current index is not −1, we check whether the selected task is the current task; otherwise, if the condition holds, we continue executing the current task, or just invoke the context switch procedure to swap out the current task and, then, execute the one with higher priority. The RR algorithm serves only the tasks with the same priority, which is actually the highest in the system. Each task is equally served to run on the processor until its time quantum is totally consumed. There are three points in time at which the scheduler is invoked, including when a new task arrives, when a task is terminated, and when the time quantum of the current task runs out. The context switch is only invoked at two points in time. One is when there is a new task with a priority higher than that of the current task, and the other is when the time quantum of the current task is totally consumed.

7.2 Memory Management

The design and implementation of *memory management* consist of two main parts: memory layout and heap management. The *memory layout* describes the memory space allocation for the RTOS kernel, the user tasks, and the boot sector.

As shown in Fig. 7.4, in the memory layout, we partitioned the main memory into several regions. The default size of the main memory is 8 MB. The region of the boot sector occupies 512 bytes and the size of the kernel context is 2 MB. The kernel context comprises the interrupt vector, kernel code, kernel data, kernel heap, kernel stack, and the system call code. The size of the kernel stack in a user task supports at least 256 recursive function calls, which is standard in C language. Correspondingly, the size of the interrupt vector supports 256 ISRs. In our implementation, at most five tasks can be simultaneously inserted into the main memory for execution and scheduling. The context of a user task comprises the code, data, heap and stack, and the size of the context is 775 KB.

The main goal of the *heap management* is to provide the kernel or user tasks with the capability of dynamic allocation of memory. In our implementation, we adapt the buddy memory allocation technique [93] to manage the heap of kernel and user tasks. Compared to the memory allocation technique used in conventional operating systems, the buddy memory allocation technique is easy to implement and can work well without using the *memory management unit* (MMU) not supported by the PLX processor. This capability makes our RTOS for PLX much easier to port to other target platforms.

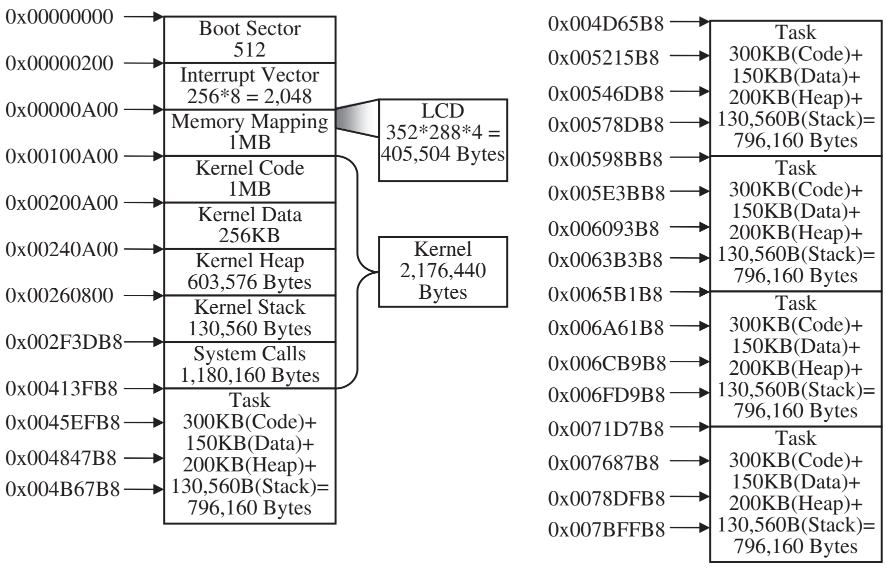

0x00000000
Boot Sector 512
0x00000200
Interrupt Vector 256*8 = 2,048
0x00000A00
Memory Mapping 1MB
LCD 352*288*4 = 405,504 Bytes
0x00100A00
Kernel Code 1MB
0x00200A00
Kernel Data 256KB
0x00240A00
Kernel Heap 603,576 Bytes
Kernel 2,176,440 Bytes
0x00260800
Kernel Stack 130,560 Bytes
0x002F3DB8
System Calls 1,180,160 Bytes
0x00413FB8
Task 300KB(Code)+ 150KB(Data)+ 200KB(Heap)+ 130,560B(Stack)= 796,160 Bytes
0x0045EFB8
0x004847B8
0x004B67B8
0x004D65B8
0x005215B8
0x00546DB8
0x00578DB8
Task 300KB(Code)+ 150KB(Data)+ 200KB(Heap)+ 130,560B(Stack)= 796,160 Bytes
0x00598BB8
0x005E3BB8
0x006093B8
0x0063B3B8
Task 300KB(Code)+ 150KB(Data)+ 200KB(Heap)+ 130,560B(Stack)= 796,160 Bytes
0x0065B1B8
0x006A61B8
0x006CB9B8
0x006FD9B8
Task 300KB(Code)+ 150KB(Data)+ 200KB(Heap)+ 130,560B(Stack)= 796,160 Bytes
0x0071D7B8
0x007687B8
0x0078DFB8
0x007BFFB8
Task 300KB(Code)+ 150KB(Data)+ 200KB(Heap)+ 130,560B(Stack)= 796,160 Bytes

Fig. 7.4 Memory layout in RTOS for PLX

Besides the above advantages, the buddy memory allocation technique has little external fragmentation, and the overhead of compacting memory is very small. However, the disadvantage of the adopted technique is a moderate amount of internal fragmentation. For example, when a request of nine memory blocks is required, then the buddy memory allocation technique allocates sixteen memory blocks to satisfy the request. Seven memory blocks are, thus, wasted. The number of allocated blocks must be a multiple of 2. The size of an allocated basic memory block is 512 bytes. The allocated blocks for each request have to be continuous; thus, it is convenient for the user to access the allocated memory in an integral and continuous way.

7.3 Communication and Synchronization Primitives

Being a multi-tasking system, an application may be divided into several tasks that collaborate to accomplish the functions of the applications. Thus, in our RTOS we provide some task communication primitives. Sharing resources among tasks can be dangerous in a multi-tasking and time-sharing system, when there is no synchronization primitive to serialize these cooperative tasks which concurrently access the same shared resources, in particular, when the shared memory is used for data exchange or communication.

The implementation of shared memory primitives is based on heap management. When the system call for creating a shared memory is invoked, the kernel allocates the memory blocks via our buddy memory allocation system, and the shared memory blocks are allocated with a kernel heap. Each shared memory is assigned a key by the user. When the key does not exist in our system, the kernel creates a proper size of memory blocks for sharing and associates the key to the shared memory. If the key does exist, that means there exists a shared memory with the same key associated to it, and the kernel returns the start address of the existing shared memory. The tasks which communicate via the same shared memory can get the shared start address through the same key related to the shared memory.

In our implementation the kernel keeps some important data structures to track the use of a shared memory. The data structure contains the key associated to the shared memory and the start address of the shared memory blocks. When all tasks release their shared memory, the kernel also returns the allocated memory to the buddy memory allocation system.

Communication or data exchange among tasks through a shared memory is widely used in most operating systems. However, concurrent accesses to shared resources could result in data inconsistency in the shared memory. In order to cope with this problem, each task has to be synchronized to access the shared data. Thus, a synchronization primitive is also provided by our RTOS for PLX.

The semaphore, a classic synchronization primitive, is often applied to protect variables or abstract data types. Using a semaphore to limit access to shared resources in a multi-tasking system is pervasively used in the OS kernel and user tasks. A task can access a shared resource only through a semaphore as follows.

When a task enters its critical section (CS), no other tasks can enter this critical section. The shared resources that can be accessed only in the critical section are thus protected. The `mutex` is a shared variable of a semaphore, each task tries to lock the `mutex` before it enters the CS, and when a task leaves the CS, the task unlocks the `mutex`. It is universally acknowledged that the `mutex` itself is also a contention among the tasks using the same semaphore. In our implementation of semaphore, when a task tries to lock the `mutex,` the kernel masks all the interrupts in PLX in order to prevent the preemption of the task; hence, the lock operation to the `mutex` is performed atomically.

There are two types of semaphores, namely binary semaphore and counting semaphore. In the binary semaphore the value of the mutex can only be zero or one, but in the counting semaphore the value of the mutex can be any integer. We only implement the binary semaphore in our RTOS because it is simple and the counting semaphore can be easily implemented using the binary semaphore.

7.4 Multimedia Applications in RTOS for PLX

The design of the PLX processor has provided not only general-purpose instructions for the normal applications, but also multimedia extension instructions for the multimedia applications to speedup their performance. However, without appropriate scheduling among the applications, it may lead to bad throughput or may not meet the required constraints of some particular applications. In a general-purpose operating system [94], the strategies of a scheduler are often to support multiple objectives such as fast response time for interactive applications and high throughput for batch applications. Though the general-purpose scheduler can meet the requirements of most applications, its inherent capability of supporting particular applications may not be sufficient enough for some special objectives, such as meeting real-time constraints.

Today multimedia applications are pervasively seen in personal computers and embedded systems, such as handheld devices. Multimedia data is often stored in a file system and can be transmitted through the network. Some multimedia services, such as digital multimedia on demands, continuously transmit multimedia data from servers to clients. The transmission of multimedia data is called *streaming*. Soft real-time streaming delivers a portion of multimedia data on the network to be played back at a client, while maintaining a desired *quality of service* (QoS). A typical example of QoS is, when a client requests a video stream from a server, it is desired that the video must be continuously displayed at 24–30 frames per second on average. To meet the soft real-time constraints of multimedia streaming, not only must the bandwidth of the network be enough for downstreaming, but the server also has to provide sufficient throughput to satisfy the multiple requests from multiple clients.

Most multimedia data streaming on the network is compressed from raw data through a compression procedure. The reduced size of the compressed data is suitable for network streaming. However, most of the compression procedures are computing-intensive, such as the H.264 data encoding process [95] as shown in Fig. 2.15; thus, with the inherent capability of the PLX processor, the provided multimedia extension instructions can meet the requirements of computing-intensive applications. However as mentioned before, without a proper scheduler, it is difficult to guarantee the QoS of multimedia streaming. As discussed in Section 7.1, the algorithm of our PRRP scheduler selects the task with the highest priority to execute, and if there is more than one ready task with the higher priority, the RR algorithm is used to schedule those tasks using FIFO. When an application needs to meet real-time constraints, we can assign a high priority to the tasks of that application, for example as shown in Fig. 2.15, DCT, IDCT, Quantization, and Entropy encoder are assigned a high priority, such that the application can gain more computation time from the system and increase its throughput to the targets. Due to the RR scheduling algorithm, the response time of a high priority task in different applications is also shortened. As a result, clients can retrieve a part of their respective multimedia data simultaneously within a small time quantum, hence, the requirement of soft real-time QoS constraints is satisfied by our RTOS through the PRRP scheduling.

There may also be data dependencies among tasks in an application. The data compression flow of H.264 illustrates that the tasks, DCT and Quant, have data dependency, meaning that these two tasks need to communicate. The inter-task communication and data consistency can be achieved through the communication and synchronization primitives implemented in our RTOS as described in Section 7.3.

7.5 Application Development Environment

To develop an application based on the RTOS for PLX requires several design tools. In this section, we introduce the following development tools and the simulation environment for building applications based on our RTOS for PLX.

(1) LCC Compiler,
(2) PLX Assembler,
(3) Parser/Locator/Loader,
(4) GNU make,
(5) Linux Distribution, and
(6) PLX Instruction Set Simulator.

Using the above tools, we show how to build the overall development environment and the toolchain. The main development environment includes the compiler, the assembler, the platform simulator as shown in Fig. 7.5, and the parser and locator as shown in Fig. 7.1. In Fig. 7.5, The PLX v1.1 LCC compiler was developed by the PALM group of Princeton University led by Professor Ruby Lee. The PLX v1.2 ASM assembler and the SystemC-based PLX v1.2 *instruction set simulator* (ISS), as described in Sections 6.3 and 6.4, were developed by the research group at National Taiwan University led by Professor Sao-Jie Chen. Dotted lines represent the input,

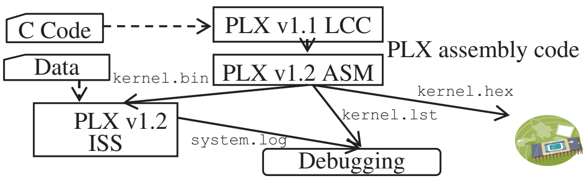


Fig. 7.5 Toolchain flow for application development on RTOS for PLX

and solid lines represent the main toolchain flow. The C code can be input to the PLX v1.1 LCC compiler, which generates PLX assembly code that is taken as input for the PLX v1.2 ASM assembler. The output of the PLX v1.2 ASM includes three files, namely `kernel.bin`, `kernel.lst`, and `kernel.hex`, which are used for simulation, debugging, and downloading, respectively.

7.5.1 Compilers

The compiler used for compiling user-written C codes is the *little C compiler* (LCC) [84], which is a lightweight compiler modified from the well-known GCC compiler and a retargetable compiler for Standard C. Several architectures, such as ALPHA, SPARC, MIPS R3000, and Intel x86 have been supported by LCC. The PALM group also modified the LCC to support the PLX processor. After compilation, the LCC produces the corresponding PLX assembly code. However, the LCC compiler was developed for the PLX v1.1 ISA. Before compiling an application using LCC, several constraints listed below must be followed.

(1) Applications must be written in ISO-C.
(2) Multiple include files are not supported, so all prototypes must be integrated into a single include file.
(3) Floating point operations are not supported, thus only fixed-point integer operations can be used in user-given C code. And
(4) I/O related standard C library is currently not supported; only the `printf` function can be used.

Users who want to use LCC must first setup a Unix-like operating system as the execution environment. To install the LCC compiler, the source file for LCC can be downloaded from [96]. The `readme` file from the archive contains an installation guide.

7.5.2 Parser, Locator, Loader, and Startup Code

The second component of the toolchain consists of a parser, a locator, and a loader. The main functions of these tools include parsing and locating PLX assembly code and loading machine code into memory. Locating involves the assignment of the proper physical memory addresses to symbols according to the memory layout as

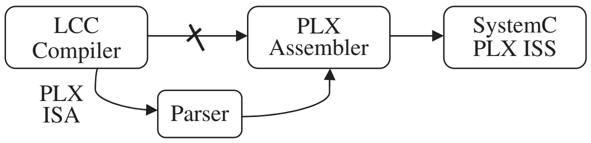


Fig. 7.6 Toolchain incompatibilities solved by our parser

shown in Fig. 7.4. Parsing tries to alleviate the version incompatibilities between the LCC compiler and the PLX assembler as shown in Fig. 7.6. The assembly code generated by LCC v1.1 cannot be directly fed into the PLX v1.2 assembler because of the following reasons.

(1) No memory allocation scheme for data is supported by the PLX v1.2 assembler.
(2) The `trap` and `lea` instructions are not supported by the PLX v1.2 assembler.
(3) Floating point instructions are not supported by the LCC v1.1 compiler and the PLX v1.2 assembler.

Figure 7.7 shows the relations of the parser, the loader, the locator, and the startup code. The *parser* takes charge of identifying the global and static variable offsets in code segment and data segment, as well as, recording all initial values of variables stored in the header of binary code. The *locator* computes and assigns the physical address of the main memory to the global or static variables in the code segment according to the memory layout and parser's offset information. Besides the above functions, the locator also prepends the startup code for each task to initialize their execution environment, such as the setting of the address of the frame pointer, the stack pointer and the initial value of static or global variables. Finally, the *loader* loads the binary file from the external SRAM into main memory and gives the control to the startup code.

The input file of the parser is called `kernel`, which is generated by the LCC compiler, and the output file is `kernel.asm` which satisfies the PLX assembler

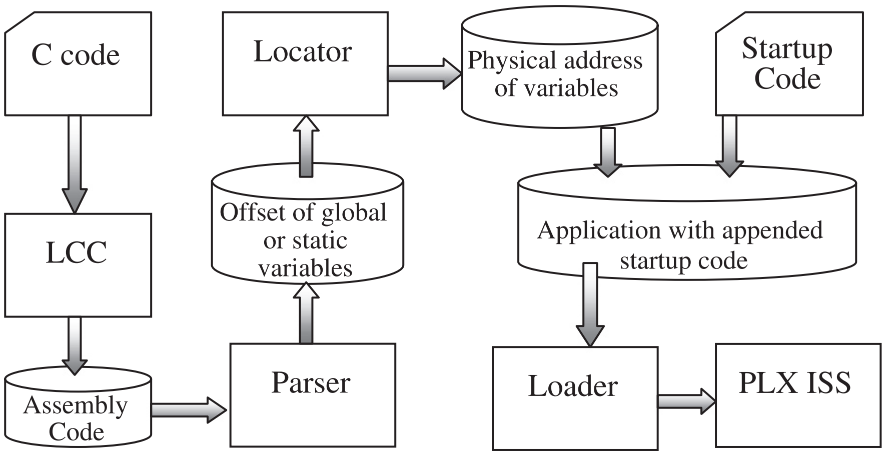

C code
Locator
Physical address of variables
Startup Code
Offset of global or static variables
Application with appended startup code
LCC
Assembly Code
Parser
Loader
PLX ISS

Fig. 7.7 Data flow among parser, locator, and Loader

input requirements. The input of the locator is the output file from the parser, namely `kernel.asm`, and the output file is an updated assembly code with a startup code.

The functionalities of the startup code include the following.

(1) To initialize the global and static variables by copying the values stored in the external SRAM,
(2) To clear all the uninitialized data (bss) section to zero,
(3) To assign the stack and frame pointers for each task, and
(4) To invoke the main function of the user task.

After the main function is invoked, the user tasks start executing.

7.5.3 PLX Platform Simulator

Currently, our RTOS for PLX is developed and tested on the instruction set simulator developed by the research group at National Taiwan University. The PLX platform simulator works on the Microsoft Windows operating system and is implemented using the SystemC library. SystemC is based on the C++ language and can be seen as a combination of hardware description language (Verilog) and software programming language. With the capability of transaction-level modeling and behavioral modeling, SystemC is also a powerful system description language. The design of the PLX platform simulator integrates several memory blocks including SRAM, embedded SRAM, external SRAM, D-cache, and I-cache. Our memory layout is based on a SRAM block of size 8 MB. The simulator also provides timers and key interrupts that are used by the RTOS.

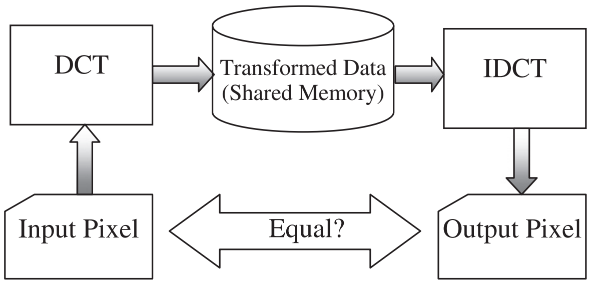
With appropriate commands to execute the simulator, we can assign the executable files or data files to the designated addresses of some memory blocks. For example, with the command, `code.bin@ICACHE:0 test.yuv@SRAM:0x10`, the simulator loads `code.bin` into 0x00000000 of I-cache and `test.yuv` into 0x30000010 of SRAM. The locator automatically generates the commands for the users to startup the simulator and to load the tasks and data to the proper addresses.

7.6 Experimental Results

In our experiments, we used two application examples to demonstrate the creation of applications and to verify our RTOS environment. The first example is a discrete transform application that is used in several popular encoding/decoding image and video standards such as JPEG and H.264. The second example is a more complex application consisting of a simplified H.264 encoding/decoding system. We chose these two applications because PLX is designed for multimedia application and these two represent the core and the state-of-the-art in multimedia applications.

The first application consists of two tasks namely *Discrete Cosine Transform* (DCT) and *Inverse Discrete Cosine Transform* (IDCT), which are pervasively used in multimedia and digital image processing, such JPEG and H.264 video encoding

Fig. 7.8 Test flow using DCT and IDCT



and decoding (refer to Section 2.3.4 for detail). The test flow is depicted in Fig. 7.8. The input to the DCT task consists of image pixel data. The output data of the DCT task consists of transformed data, which is then given as input data for the IDCT task. For verification, we compare the output data of the IDCT with the original pixel data to see if they are the same.

The 4×4 DCT is responsible for the transformation of the integer image data from the space domain to the frequency domain. In contrast to DCT, the 4×4 IDCT computes the inverse transform. Thus, the data in IDCT is transformed from the frequency domain to the space domain.

Before we start to compile the above two tasks, the LCC compiler for PLX has to be installed, which can be downloaded from [96]. The instructions in the `readme` file can be used to install the LCC compiler. After installing the LCC compiler, several built-in examples can be found at the following path: `PLXROOT/PLX-1.1/benchmarks`. Each example is contained in the directory, whose name is prefixed with "`lcc.`" You may compile these examples via the command, `run-bench -c lcc.examples`, to check if your compiler is successfully installed. Two new projects will be created in the benchmarks directory including `lcc.dct and lcc.idct.`

To compile these projects, two commands, `run-bench -c lcc.dct` and `run-bench -c lcc.idct`, must be issued. After compiling the projects, the LCC compiler generates two directories with the corresponding names of the projects in the directory, `PLXROOT/PLX-1.1/build,` where each project has a file named `kernel`. In order to distinguish from other applications, we rename the files with the same name, `kernel,` to `kernel_dct and kernel_idct` corresponding to the two projects.

In the next step, we perform the static memory allocation for the codes. To invoke the locator and to use the file, `kernel,` as the input, the following command must be issued, `./locator kernel`. The locator generates three files for each of the projects, including `kernel.asm, kernel_data.out,` and `kernel.arg`. The main purposes of these three files are described below. However, in our example, we use the two commands, `./locator kernel_dct` and `./locator kernel_idct,` and the generated files are `kernel_dct.asm, kernel_dct_data.out, kernel_dct.arg, kernel_idct.asm, kernel_idct_data.out,` and `kernel_idct.arg`.

Corresponding to the two projects, `lcc.dct` and `lcc.idct`, the generated contents of the arg files are listed as follows:

```
// filename: kernel_dct.arg
kernel_dct.bin@EXTSRAM:0 kernel_dct_data.out@EXTSRAM:0x4B000

// filename: kernel_idct.arg
kernel_idct.bin@EXTSRAM:0x70810
kernel_idct_data.out@EXTSRAM:0xBB810
```

- `kernel.asm`: In this assembly file, all incompatibilities between the LCC compiler and the assembler have been removed, and the assembly code is thus compatible to the PLX v1.2 instruction set architecture.
- `kernel_data.out`: This binary file contains the loading information and the initial values of the global and static variables in `kernel.asm.` The loading information occupies totally 16 bytes and consists of the following fields: text, data, *block started by symbol* (bss), and reserved field. Each field occupies 4 bytes. This loading information will be added in the head of the binary executable file.
- `kernel.arg`: This file contains the parameter information for issuing commands to the PLX simulator. The simulator locates the text, data, bss, and initial information to the proper locations as specified in the command line.

In the last step, we use the PLX assembler to assemble the `kernel_dct.asm` and `kernel_idct.asm` to get the binary files, namely `kernel_dct.bin` and `kernel_idct.bin`. The code, data, bss, and initial variable values corresponding to a task are placed in the external SRAM (XRAM). A dynamic loader is used here to load the content of a context into the proper address of the main memory. The dynamic loader in default is located at the address 0x0, which in memory layout is the start address of boot sector. Hence, we give the command to invoke the PLX ISS as follows:

```
''\PLX_platform Dynload.bin@SRAM:0x0
kernel_dct.bin@EXRAM:0 kernel_dct_data.out@XRAM:0x4B000
kernel_idct.bin@XRAM:0x70810 kernel_idct_data.out@XRAM:0xBB810''.
```

Figure 7.9 shows the main purpose of the command line for the simulator, that is, to load the loader, `dct` and `idct` codes, and the loading information of `dct` and `idct` to the specifically physical addresses in the designated memory block. When we want to start executing a task, we press a designated key to invoke the key interrupt service routine and the dynamic loader starts to load the content of that task from the XRAM to the main memory; subsequently, the scheduler starts to schedule the loaded task.

The two-dimensional matrix input of DCT is shown in Fig. 7.10(a), and the two-dimensional matrix output of DCT is shown in Fig. 7.10(b) which is also the input of IDCT.

Fig. 7.9 Start addresses for binary files in the DCT/IDCT example

File name	File description	Memory block	Start address
`Dynload.bin`	Loader	SRAM	00000000
`kernel_dct.bin`	The DCT binary code	XRAM	00000000
`kernel_dct_data.out`	The loading information of DCT	XRAM	0004B000
`kernel_idct.bin`	The IDCT binary code	XRAM	00070810
`kernel_idct_data.out`	The loading information of DCT	XRAM	00BB810

Fig. 7.10 Two-dimensional matrix input/output of DCT

$$\text{(a)}\begin{pmatrix} -233 & -58 & 31 & -11 \\ -27 & 29 & -18 & 8 \\ -11 & -3 & -7 & 0 \\ -2 & -1 & 0 & -1 \end{pmatrix} \quad \text{(b)}\begin{pmatrix} 50 & 54 & 60 & 65 \\ 63 & 58 & 65 & 89 \\ 61 & 59 & 68 & 113 \\ 53 & 57 & 70 & 122 \end{pmatrix}$$

We allocate a block of shared memory from the heap of the kernel for communication between the two tasks. After implementing the two tasks, DCT and IDCT, we placed them into the external memory using the simulation command. The locator and loader were used to locate and load these two codes. The RTOS scheduler successfully scheduled the two tasks in the system to run on the ISS. Finally, we compared the output data of IDCT with the original data and found that they were the same.

The H.264/AVC Video Coding Standard specifies an integer DCT-like 4×4 transform designed to be computed using only additions, subtractions and shifts on data unit. Using PLX subword parallelism instructions, like `padd`, `psub`, and `pshift`, we can tremendously reduce the complexity of computation. The same concept is also implemented in the Inverse DCT. According to the description of H.264/MPEG4 Part 10 White paper [97], H.264 uses a scalar quantization which includes large array computations. In an SIMD architecture, we can use a single PLX instruction, such as `pmul.odd` and `pmul.even`, to accomplish the multiple multiplications on multiple data units.

We grouped the four jobs, namely DCT, Quantization [98,99], Inverse DCT, and Inverse Quantization, into two tasks, one of which executes the DCT and Quantization, and the other executes Inverse DCT and Inverse Quantization. In Fig. 7.11(a), we illustrate a segment of the code for the DCT. In the 64-bit PLX processor, the four 64-bit registers, `Rc`, `Rh`, `Ri`, and `Rd`, are used to temporally store the data of 4×4 matrix, and the three operator, `padd`, `psub`, and `pshifti`, are used to compute on the four registers. For example, the instruction, `padd.2.s Re, Rc, Rd`, is used to concurrently add the 2-byte data of `Rc` and `Rd`, and the parallel

Fig. 7.11 Segment of multimedia extension instructions for (a) DCT and (b) Quantization

```
padd.2.s     Re,Rc,Rd
padd.2.s     Rj,Rh,Ri
psub.2.s     Rf,Rc,Rd
psub.2.s     Rk,Rh,Ri
pshifti.2.l  Rg,Rf,1
pshifti.2.l  Rl,Rk,1
padd.2.s     Rm,Re,Rj
padd.2.s     Rn,Rk,Rg
psub.2.s     Ro,Re,Rj
psub.2.s     Rp,Rf,Rl
```

```
pmul.odd     R4,L1,R1
pmul.even    R5,L1,R1
pmul.odd     R6,L2,R2
pmul.even    R7,L2,R2
mix.2.l      R3,R1,R2
pmul.odd     R8,L3,R3
pmul.even    R9,L3,R3
mix.2.r      R3,R2,R1
pmul.odd     R10,L4,R3
pmul.even    R11,L4,R3
```

result is stored in the `Re` register. The instruction, `psub.2.s Rf, Rc, Rd`, is similar to `padd.2.s`, and it subtracts 2-byte data of `Rc` and `Rd` and stores the result in `Rf`. The instruction, `pshifti.2.1 Rg, Rf, 1`, is to shift 1-bit to each 2-byte data of `Rf` to the left and stores the result in `Rg`.

In Fig. 7.11(b), we illustrate a segment of the code for Quantization. The registers, `L1, L2, L3` and `L4,` are used to store the results of DCT, and the registers, `R1, R2, R3` and `R4,` are used to store the multiplication factors. Due to the fact that n-bit multiplication will result in a $2n$-bit output, in PLX processor architecture, the designer divides the instruction of a multiplication into even and odd multiplications. For example, the instruction, `pmul.odd R4, L1, R1`, is to multiply the field of odd index in `L1` and `R1` and the result is subsequently stored in `R4`.

In contrast to the multiplication instruction of x86 architecture, the performance of PLX subword parallel operation is about 2 times faster than the multiplication instruction of x86. We used a raw image as the input for this example and retrieved the output, and then, we displayed the output image to see if the image is similar to the original one. The result shows that this example worked correctly.

As a consequence, with the two successful examples, we verified the parser, locater, loader, scheduler, and dispatcher of our RTOS, and also illustrated the capability of the PLX virtual prototype platform.

Chapter 8
Conclusion

We had designed a PLX-based embedded system for H.264 application. The system design was started with knowledge obtained from many multimedia applications. By analyzing multimedia applications as shown in Chapter 2, we decided that the processor needs an SWP-SIMD instruction set to process multimedia applications in a data level parallelism way. By performance analysis in Section 4.1, we know that a native SWP-SIMD PLX processor can handle low-resolution picture for low-cost product, but high-resolution picture needs specific motion-estimation hardware or ASIP. An un-timed virtual prototype platform and approximate-time transaction-level SystemC modeling as presented in Chapter 6 were built to develop compiler and OS in early design stage before PLX RTL circuit design is ready. Chapter 7 demonstrated our experiences in using this virtual prototype platform to develop a real-time OS for PLX.

The RTOS implementation comprises: (1) a priority-based, round-robin, preemptive PRRP scheduler that is capable of multi-programing; (2) a dispatcher that is responsible for context switch between two tasks; (3) a memory manager that allow tasks to allocate or deallocate memory blocks from the heap, and to create shared memory for communication; (4) a loader that reads codes from the external RAM and initializes the execution environment for the tasks; (5) a locator that is responsible for calculating the physical address for global or static variables; and (6) the ISRs which handle the key interrupt and timer interrupt.

With the support of multimedia extension instructions of the PLX ISS and priority-based scheduling, we can assign higher priorities to the application tasks that are constrained by time. Consequently, our scheduler is suitable for the multimedia applications to meet their real-time constraints easily, performing on PLX.

Besides the main components of the RTOS, we also introduced a toolchain for RTOS development, which comprises an LCC C compiler, an assembler, and a parser that fixes the incompatibilities between the compiler and the assembler. These tools can also help users to design their own applications to run on the RTOS for PLX. The SystemC-based ISS for PLX generates the debug information in a log file, which can be used by users to debug their applications.

In the future, we will design the I/O device drivers for more peripherals, such as the hard disk and the network interface, which will provide application developers with a more powerful environment for different applications.

For the cost/power efficiency and programmability purposes, a system-on-chip embedded with an application-specific instruction set processor is necessary at the nano-meter era. A complicated hardware-in-the-loop design flow induces heavy work on system-level hardware/software codesign engineers who have to develop system level models, a multi-level parallelized processor, a parallel compiler, and a real-time OS for a many-core SoC. This book tried to give an overall introduction to materials on all these knowledge domains. The experiences gained in the implementation of a PLX processor and toolochain design show that we can use these system-level design and verification tools to shorten the design cycle. We hope that our success story will encourage readers to develop their own SoCs with an in-house single-core or multi-core processor inside.

References

1. R. B. Lee and A. M. Fiskiran, “PLX: An Instruction Set Architecture and Testbed for Multimedia Information Processing,” *Journal of VLSI Signal Processing*, vol. 40, no. 1, pp. 85–108, May 2005.
2. R. B. Lee and A. M. Fiskiran, “PLX: a Fully Subword-Parallel Instruction Set Architecture for Fast Scalable Multimedia Processing,” *Proceedings of IEEE International Conference on Multimedia and Expo*, pp. 117–120, Aug. 2002.
3. A. Sangiovanni-Vincentelli and G. Martin, “Platform-Based Design and Software Design Methodology for Embedded Systems,” *IEEE Design & Test of Computers*, vol. 18, no. 6, pp. 23–33, Nov/Dec 2001.
4. Texas Instruments, http://focus.ti.com/general/docs/wtbu/wtbuproductcontent.tsp?contentId=4666&navigationId=12314&templateId=6123.
5. R. Zurawski and M.-C. Zhou, “Petri Nets and Industrial Applications: A Tutorial,” *IEEE Transactions on Industrial Electronics*, vol. 41, no. 6, pp. 567–583, Dec. 1994.
6. D. D. Gajski, N. D. Dutt, A. C.-H. Wu, and S. Y.-L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic, 1992.
7. S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, “Coordinated Parallelizing Compiler Optimizations and High-Level Synthesis,” *ACM Transactions on Design Automation of Electronic Systems*, vol. 9, no. 4, pp. 441–470, Oct. 2004.
8. J. Ferrante, K. J. Ottenstein and J. D. Warren, “The Program Dependence Graph and its Use in Optimization,” *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 3, pp. 319–349, Jul. 1987.
9. S. Horwitz, T. Reps, and D. Binkley, “Interprocedural Slicing Using Dependency Graphs,” *ACM Transactions on Programming Languages and Systems*, vol. 22, no. 1, pp. 26–60, Jan. 1990.
10. A. Tamhankar and K. R. Rao, “An Overview of H.264/MPEG-4 Part 10,” *Proceedings of the 4th EURASIP Conference on Video/Image Processing and Multimedia Communications*, EC-VIP-MC 2003, pp. 1–51, Jul. 2003.
11. S. Y. Yap and J. V. McCanny, “A VLSI Architecture for Variable Block Size Video Motion Estimation,” *IEEE Transactions on Circuits and Systems*, vol. 51, no. 7, pp. 384–389, Jul. 2004.
12. L. Deng, W. Gao, M. Z. Hu, and Z. Z. Ji, “An Efficient Hardware Implementation for Motion Estimation of AVC Standard,” *IEEE Transactions on Consumer Electronics*, vol. 51, no. 4, pp. 1360–1366, Nov. 2005.
13. C. M. Ou, C. F. Lee, and W. J. Hwang, “An Efficient VLSI Architecture for H.264 Variable Block Size Motion Estimation,” *IEEE Transactions on Consumer Electronics*, vol. 51, no. 4, pp. 1291–1299, Nov. 2005.
14. T. P. Chen, H. Haussecker, A. Bovyrin, R. Belenov, K. Rodyushkin, A. Kuranov and V. Eruhimov, “Computer Vision Workload Analysis: Case Study of Video Surveillance Systems,” *Intel Technology Journal*, vol. 9, no. 2, pp. 109–118, May 2005.

15. F. Russo and A. Lazzari, “Color Edge Detection in Presence of Gaussian Noise Using Nonlinear Prefiltering,” *IEEE Transactions on Instrumentation and Measurement*, vol. 54, no. 1, pp. 352–358, Feb. 2005.
16. F. M. Alzahrani and T. Chen, “A Real-Time Edge Detector: Algorithm and VLSI Architecture,” *Real-Time Imaging*, vol. 3 no. 5, pp. 363–378, Oct. 1997.
17. A. Broggi, M. Bertozzi, A. Fascioli, C. G. Lo Bianco, and A. Piazzi, “Visual Perception of Obstacles and Vehicles for Platooning,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 01, no. 3, pp. 164–176, Sep. 2000.
18. M. B. van Leeuwen and F. C. A. Groen, “Vehicle Detection with a Mobile Camera: Spotting Midrange, Distant, and Passing Cars,” *IEEE Robotics and Automation Magazine*, vol. 12, no. 1, pp. 37–43, Mar. 2005.
19. Z. Sun, G. Bebis, and R. Miller, “On-Road Vehicle Detection Using Evolutionary Gabor Filter Optimization,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 6, no. 2, pp. 125–137, Jun. 2005.
20. S. Gupte, O. Masoud, R. F. K. Martin, and N. P. Papanikolopoulos, “Detection and Classification of Vehicles,” *IEEE Transactions on Intelligent Transportation Systems*, vol. 3, no. 1, pp. 37–47, Mar. 2002.
21. P. L. Montgomery, “Modular multiplication without trial division,” *Mathematics of Computation*, vol. 44, no. 170, pp. 519–521, Apr. 1985.
22. http://en.wikipedia.org/wiki/DES
23. http://en.wikipedia.org/wiki/AES
24. http://www.systemc.org/groups
25. N. D. Liveris, H. Zhou, and P. Banerjee, “An Efficient System-Level to RTL Verification Framework for Computation-Intensive Applications,” *Proceeding of the 14th Asian Test Symposium*, pp. 28–33, Dec. 2005.
26. J. R. Harrison and L. Théry, “A skeptic’s Approach to Combining HOL and Maple,” *Journal of Automated Reasoning*, vol. 21, no. 3, pp. 279–294, Dec. 1998.
27. B. Akbarpour and S. Tanar, “An Approach for the Formal Verification of DSP Design Using Theorem Proving,” *IEEE Transactions on Computer-Aided Design*, vol. 25, no. 8, pp. 1441–1457, Aug. 2006
28. E. Clarke, D. Kroening, and K. Yorav, “Behavioral Consistency of C and Verilog Programs Using Bounded Model Checking,” *Proceedings of Design Automation Conference*, pp. 368–371, Jun. 2003.
29. A. Jerraya, J. Borel, A. Sauer, W. Rosenstiel, F. Ghenassia, and E. Perea, “Design Automation in Europe,” *IEEE Design & Test of Computers*, vol. 16, no. 4, pp. 90–95, Oct/Dec 1999.
30. M. Strik, A. Gonier and P. Williams, “Subsystem Exchange in a Concurrent Design Process Environment,” *Proceedings of Design, Automation and Test in Europe Conference*, pp. 953–958, Mar. 2008.
31. A. Clouard, K. Jain, F. Ghenassia, L. Maillet-Contoz, and J. P. Strassen, “Using Transactional Level Models in a SoC Design Flow,” in *SystemC Methodologies and Applications*, Chapter 2, pp. 29–63, Ed. W. Müller, W. Rosentiel, and J. Ruf, Kluwer Academic Publishers, 2003.
32. W. Rosenstiel, S. Swan, F. Ghenassia, P. Flake, and J. Srouji, “SystemC and SystemVerilog: Where Do They Fit? Where Are They Going?” *Proceedings of Design, Automation and Test in Europe Conference and Exhibition*, vol. 1, pp. 122–127, Feb. 2004.
33. A. Hoffmann, O. Schliebusch, A. Nohl, G. Braun, O. Wahlen, and H. Meyr, “A Methodology for the Design of Application Specific Instruction-Set Processors Using the Machine Description Language LISA,” *Proceedings of the 2001 IEEE/ACM International Conference on Computer-Aided Design*, pp. 625–630, Nov. 2001.
34. A. Hoffmann, H. Meyr, and R. Leupers, *Architecture Exploration for Embedded Processors with LISA*, Kluwer Academic Publishers, 2002.
35. O. Wahlen, *C Compiler Aided Design of Application-Specific Instruction-Set Processors Using the Machine Description Language LISA*, Shaker Verlag, 2004.

36. CoWare Inc., *LISATek Software Development Tools Manual – 2005.1.0.*
37. http://www.ace.nl/compiler/cosy-express.html
38. G. H. Barnes, R. M. Brown, M. Kato, D. J. Kuck, D. L. Slotnick and R. A. Stokes, "The ILLIAC IV Computer," *IEEE Transactions on Computers*, vol. C-17, no. 8, pp. 746–757, Aug. 1968.
39. R. M. Russel. "The CRAY-1 Computer System," *Communications of ACM*, vol. 21, no. 1, pp. 63–72, Jan. 1978.
40. R. B. Lee, "Accelerating Multimedia with Enhanced Microprocessors," *IEEE Micro*, vol. 15, no. 2, pp. 22–32, Apr. 1995.
41. A. Peleg and U. Weiser, "MMX Technology Extension to the Intel Architecture," *IEEE Micro*, vol. 16, no. 4, pp. 42–50, Aug. 1996.
42. Intel Corporation, *IA32 Intel Architecture Software Developer's Manual*, http://www.intel.com/products/processor/manuals/
43. Sam Fuller, *Motorola's Altivec technology*, Freescale 1998, http://www.freescale.com/files/32bit/doc/fact_sheet/ALTIVECWP.pdf
44. *3Dnow! Technology Manual*, Advanced Micro Devices, Inc., 1999, http://www.amd.com
45. M. Tremblay, J. M. O'Connor, V. Narayanan, and H. Liang, "VIS Speeds New Media Processing," *IEEE Micro*, vol. 16, no. 4, pp. 10–20, Aug. 1996.
46. *MIPS Extension for Digital Media with 3D*, MIPS Technologies, Inc., Mar. 1997, http://www.mips.com
47. Y. Cao and H. Yasuura, "A System-Level Energy Minimization using Datapath Optimization," *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 231–236, Aug. 2001.
48. T. Ishihara and H. Yasuura, "Programmable Power Management Architecture for Power Reduction," *IEICE Transactions on Electronics*, vol. E81-C no. 9, pp. 1473–1480, Sep. 1998.
49. A. Sinha, A. Wang, and A. P. Chandrakasan, "Algorithmic Transforms for Efficient Energy Scalable Computation," *Proceedings of the International Symposium on Low Power Electronics and Design*, pp. 31–36, Jul. 2000.
50. G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The Microarchitecture of Pentium 4 Processor," *Intel Technology Journal*, vol. 5, no. 1, pp. 13–24, Feb. 2001.
51. http://focus.ti.com/docs/prod/folders/print/tms320c6410.html
52. D. D. Gajski, *NISC: The Ultimate Reconfigurable Component*, Center for Embedded Computer Systems Technical Report CECS TR 03–28, University of California, Irvine, October 1, 2003.
53. D. A. Patterson, "Reduced Instruction Set Computers," *Communications of ACM*, vol. 28, no. 1, pp. 8–21, Jan. 1985.
54. T. Jamil, "RISC versus CISC," *IEEE Potentials*, vol. 14, no. 3, pp. 13–16, Aug.–Sep. 1995.
55. D. M. Tullsen and J. A. Brown, "Handling Long-Latency Loads in a Simultaneous Multithreading Processor," *Proceedings of the 34th International Symposium on Microarchitecture*, pp. 318–327, Dec. 2001.
56. D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller and M. Upton, "Hyper-Threading Technology Architecture and Microarchitecture," *Intel Technology Journal*, vol. 6, no. 1, pp. 36–46, Feb. 2002.
57. http://www.gpgpu.org/
58. http://www.nvidia.com/
59. E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J.Babb, S. Amarasinghe, and A. Agarwal, "Baring it all to Software: Raw Machines," *IEEE Computer*, pp. 86–93, Sep. 1997.
60. M. Azimi, N. Cherukuri, D. N. Jayasimha, A. Kumar, P. Kundu, S. Park, I. Schoinas, A. S. Vaidya, "Integration Challenges and Tradeoffs for Tera-scale Architectures," *Intel Technology Journal*, vol. 11, no. 3, pp. 173–184, Aug. 2007.
61. M. B. Taylor, W. Lee, J. E. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, J. Kim, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and

A. Agarwal. “Evaluation of the Raw Microprocessor: An Exposed-Wire-Delay Architecture for ILP and Streams,” *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pp. 2–13, Jun. 2004.

62. D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, “ Dependence Graphs and Compiler Optimizations,” *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 207–218, Jan. 1981.
63. K. Kennedy and R. Allen, “Automatic Translation of FORTRAN Programs to Vector Form,” *ACM Transactions on Programming Languages and Systems*, vol. 9, no. 4, pp. 491–554, Oct. 1987.
64. M. E. Wolf and M. S. Lam, “A Loop Transformation Theory and an Algorithm to Maximize Parallelism,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 2, no. 4, pp. 452–471, Oct. 1991.
65. A. Darte and F. Vivien, “A Classification of Nested Loops Parallelization Algorithms,” *Proceedings of the IEEE Symposium on Emerging Technologies and Factory Automation*, vol. 1, pp. 217–234, Oct. 1995.
66. J. R. Allen, K. Kennedy, C. Porterfield and J. Warren, “Conversion of Control Dependence to Data Dependence,” *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 177–189, Jan. 1983.
67. R. Kramer, R. Gupta and M. L. Soffa, “The Combining DAG: a Technique for Parallel Data Flow Analysis,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 8, pp. 805–813, Aug. 1994.
68. R. Tarjan, “Depth-first Search and Linear Graph Algorithms,” *SIAM Journal on Computing*, vol. 1, no. 2, pp. 146–160, 1972.
69. A. E. Eichenberger, P. Wu, and K. O’Brien, “Vectorization for SIMD Architectures with Alignment Constraints,” *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pp. 82–93, Jun. 2004.
70. P. Wu, A. E. Eichenberger, and A. Wang, “Efficient SIMD Code Generation for Runtime Alignment and Length Conversion,” *Proceedings of International Symposium on Code Generation and Optimization*, CGO, pp. 153–164, Mar. 2005.
71. G. Ren, P. Wu, and D. Padua, “Optimizing Data Permutations for SIMD Devices,” *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 118–131, Jun. 2006.
72. S. Larsen and S. Amarasinghe, “Exploiting Superword Level Parallelism with Multimedia Instruction Sets,” *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pp. 145–156, Jun. 2000.
73. F. Franchetti, S. Kral, J. Lorenz, and C. W. Ueberhuber, “Efficient Utilization of SIMD Extensions,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 409–425, Feb. 2005.
74. S. Larsen, R. Rabbah, and S. Amarasinghe. “Exploiting Vector Parallelism in Software Pipelined Loops,” *Proceedings of the 38th International Symposium on Microarchitecture*, pp. 119–129, Nov. 2005.
75. D. M. Lavery and W. M. Hwu, “Modulo Scheduling of Loops in Control-Intensive Non-Numeric Programs,” *Proceedings of the 29th Annual International Symposium on Microarchitecture*, pp. 126–137, Dec. 1996.
76. G. H. Lin, S. J. Chen, R. B. Lee, and Y. H. Hu, “Memory Access Optimization of Motion Estimation Algorithms on a Native SIMD PLX Processor,” *Proceedings IEEE Asia-Pacific Conference on Circuits and Systems*, pp. 567–570, Dec. 2006.
77. S. Ryoo, S-Z Ueng, C. Rodrigues, R. Kidd, M. Frank, and W-M Hwu, “Automatic Discovery of Coarse-Grained Parallelism in Media Applications,” *Lecture Notes in Computer Science*, Springer, vol. 4050, pp. 194–213, Jan. 2007.
78. P. Tu and D. Padua, “Gated SSA-based Demand-Driven Symbolic Analysis for Parallelizing Compilers,” *Proceedings of the International Conference on Supercomputing*, pp. 414–423, Jul. 1995.
79. T. G. Mattson, B. A. Sanders, and B. L. Massingill, *Patterns for Parallel Programming*, Addison Wesley, 2005.

80. A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools*, 2nd ed., Addison Wesley, 2007.
81. http://en.wikipedia.org/wiki/Abstract_syntax_tree
82. A. I. Holub, *Compiler Design in C*, Prentice Hall, 1990.
83. http://en.wikipedia.org/wiki/Retargetable_compiler
84. D. R. Hanson and C. W. Fraser, *A Retargetable C Compiler: Design and Implementation*, Addison-Wesley, 1995.
85. GNU Compiler Collection, http://gcc.gnu.org/
86. R. Wilson, R. French, C. Wilson, S. Amarasinghe, J. Anderson, S. Tjiang, S.-W. Liao, C.-W. Tseng, M. Hall, M. Lam, and J. Hennessy, "SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers," ACM SIGPLAN Notices, vol. 29, no. 12, pp. 31–37, Dec. 1994.
87. G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis, *An Overview of the SUIF2 Compiler Infrastructure*, Computer Systems Laboratory, Stanford University, http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/
88. G. Aigner, A. Diwan, D. L. Heine, M. S. Lam, D. L. Moore, B. R. Murphy, and C. Sapuntzakis, *The SUIF Program Representation*, Computer Systems Laboratory, Stanford University, 1999, http://suif.stanford.edu/suif/suif2/doc-2.2.0-4/
89. http://www.gelato.org
90. P. P. Chang, S. A. Mahlke, W. Y. Chen, N. J. Water, and Wen-mei W. Hwu, "IMPACT: An Architectural Framework for Multiple-Instruction-Issue Processors," *Proceedings of the 18th Annual International Symposium on Computer Architecture*, pp. 266–275, May 1991.
91. http://www.gelato.uiuc.edu/OpenIMPACT-documentation.php
92. M. Samek and R. Ward, "Build a Super Simple Tasker," *Embedded System Design*, Jul. 2006, http://www.embedded.com
93. D. Knuth, *The Art of Computer Programming Volume 1: Fundamental Algorithms*, 2nd Ed., pp. 435–455, Addison-Wesley, 1997.
94. A. Silberschatz, G. Gagne, and P. B. Galvin, *Operating System Concepts (Windows XP Update)*, 6th Ed., Addison-Wiley, 2002.
95. T. Wiegand, G. J. Sullivan, G. Bjontegaard, and A. Luthra, "Overview of the H.264/AVC Video Coding Standard," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 560–576, Jul. 2003.
96. *RTOS for PLX*, Embedded Systems Laboratory, Department of Computer Science & Information Engineering, National Chung Cheng University, Chiayi, Taiwan, Jul. 2008, http://www.cs.ccu.edu.tw/∼pahsiung/plx_os/
97. I. E. G. Richardson, Transform and Quantization: H.264/MPEG4 Part 10, White Paper, Mar. 2003.
98. H. Malvar, A. Hallapuro, M. Karczewicz, and L. Kerofsky, "Low-Complexity Transform and Quantization in H.264/AVC," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 13, no. 7, pp. 598–603, Jul. 2003.
99. *H.264/AVC Reference Software*, 2008, http://iphome.hhi.de/suehring/tml/download/